MERIDIAN
SOFTWARE SYSTEMS

Meridian Ada 4.1

DOS Environment Library
User's Guide

Table of Contents

Chapter 1 Introduction

The *Meridian Ada DOS Environment Library* is a set of Ada packages for use with the Meridian Ada* compiler that provides an interface to PC–DOS version 2.1 or later. The *MeridianAda DOS Environment Library* allows you to use most of the PC–DOS system calls, plus screen management and other BIOS functions. Excluded system calls include those that are obsolete in the newer versions of PC–DOS or are designed for use only on PC networks.

Note that the BIOS interface functions are designed to operate with the standard IBM PC BIOS.

Some operations are appropriate only for DOS 3.0 or later; these are marked as such. Any attempt to use these operations on earlier versions of DOS causes an exception to be raised.

Some of the highlights of the *MeridianAda DOS Environment Library* include:

Package	Function
`absolute_disk`	Provides procedures to read and write on disk devices without regard to the file system structure imposed by PC–DOS.
`box`	Provides the procedures for drawing boxes on the text screen.
`common_display_types`	Contains type declarations for the various packages that handle display operations.
`cursor`	Provides operations to move a text mode cursor, to get the current cursor coordinates, and to alter the visibility or form of the cursor.
`directory`	Provides subprograms that create, delete, and change directories.
`disk`	Provides operations such as flushing file buffers, getting default disk drive ID, and returning information about the disk drive.
`disk_types`	Contains type declarations used by the disk and directory operation packages.
`equipment`	Provides a function that enumerates the system–recognized devices and facilities present on the machine.
`errors`	Provides PC–DOS status declarations and for version 3.0 or later of PC–DOS, a procedure to get extended status information.
`file_io`	Provides input/output, file attribute manipulation, and wildcard search operations.
`interrupt`	Allows calls to the interrupt vectors.
`memory`	Provides functions to allocate and deallocate memory.
`port`	Allows byte or word input and output to the specified port.
`program_control`	Provides functions that exert control over the executing program that execute other programs, or that obtain information about resident programs.

*The *DOS Environment Library* can also be used with the AdaGraduate Compiler. Please note that references to the Meridian Ada Compiler also apply to the AdaGraduate Compiler.

Package	Function
`revision`	Provides a facility for determining the revision number of PC–DOS.
`time`	Provides procedures to get and set the current system date and time.
`tty`	Provides operations on the console terminal display and keyboard.
`video`	Provides various output and control functions for the Monochrome, Color Graphics and Extended Graphics Adapter Cards.

1.1 Scope of This Document

It is assumed that you are familiar with PC–DOS and the Meridian Ada compiler and associated library management tools. This document covers only material required to use the packages; information about the Ada programming language, about most low–level functional aspects of the PC–DOS system calls, and most aspects of the Meridian Ada compiler is outside the scope of this document. Other documents that cover the aforementioned material include:

- The *Meridian Ada Compiler User's Guide.* This document describes operation of the compiler and associated tools.
- *Reference Manual for the Ada Programming Language ANSI/MIL–STD–1815A* (the LRM). This document describes the Ada language.
- *Ada For Programmers* (Prentice–Hall, 1983). This book by Meridian co–founders Eric Olsen and Stephen Whitehill provides an introduction to Ada programming for practicing programmers.
- *Microsoft MS–DOS Operating System Programmer's Reference* (available from Microsoft Corporation). This document covers DOS system calls in great detail.

1.2 Document Organization

The descriptions of the packages are organized alphabetically by package name. Each package is presented with its specification, a brief description of its uses, and a discussion of the details of the package.

Chapter 2 Installing the DOS Environment Library

2.1 Installation Procedure

Prior to installing the *Meridian Ada DOS Environment Library*, the Ada compiler must be installed.

This installation procedure assumes that you will install the *DOS Environment Library* on the same hard disk drive and in the same top–level directory in which you installed the compiler. If you do not install the *DOS Environment Library* in that place (not recommended) the installation procedure will work to a point, but the link entry in the *DOS Environment Library* library database file (**ada.lib**) will have to be linked to the standard distribution library (**paclib\ada.lib**). An example of how to do this is given in section 2.2.

To perform the installation procedure:

1. Reboot the operating system.
2. Insert the distribution diskette into diskette drive **a:**.
3. Run the installation program as:

   a:install *d:\directory*

   The command line arguments to **install** are *d*, a hard disk drive letter, and *directory*. Note that *directory* must start with a backslash ("\"). The hard disk and directory that you select should be the same as were selected for installation of the compiler (e.g. **c:\ada**).

The installation procedure should take only a few minutes. If there is not enough space on the destination disk to accommodate all the files, the installation will fail. The files are installed in a sub–directory named **dosenv** below the top–level directory given to the **install** command. If **dosenv** does not exist on the destination disk, then the **install** command creates that directory. If the directory already exists, a harmless error message is printed, but installation proceeds.

2.2 Adjusting an Unusual Installation

If you did not install the *DOS Environment Library* using the same top–level directory in which you installed the compiler, you should have gotten a warning message. In this case, the link entry in the *DOS Environment Library* library database file (**ada.lib**) must to be modified to reflect the actual location of the standard distribution library (**paclib\ada.lib**).

You do not need to do this if you used the same top–level directory in which you installed the compiler.

The adjustment to the *DOS Environment Library* library is made by using the **lnlib** command, as in this example:

```
e:
cd \stuff\dosenv
rem -- The above two commands assume that e:\stuff was
rem -- the top-level directory specified to the install command.
lnlib -r c:\ada\paclib\ada.lib e:\ada\paclib\ada.lib
```

This example assumes that you originally installed the compiler using **e:\ada** as the installation directory and that you installed the *DOS Environment Library* in **e:\stuff**. Note that the **lnlib** command in this example replaces a pre–existing library link entry.

2.3 Creating Library Links

Once the software is installed, it is necessary to make the appropriate link in each local library where the package is to be used. If the software is installed in the same top–level directory in which the compiler was installed, then the installation procedure automatically modifies the behavior of the **newlib** command so that the *DOS Environment Library* library is linked into every newly created local library. If the software is installed elsewhere and you want the same modification made to the **newlib** command, then you should perform the following procedures.

Modify the **newlib.bat** file to link the *DOS Environment Library* library whenever the **newlib** command is invoked. For example, this command could be added to the end of the **newlib.bat** file:

```
lnlib c:\ada\dosenv\ada.lib
```

The **newlib.bat** file is located in the **bin** directory below the Meridian Ada compiler installation directory. In this example, it is assumed that **c:\ada** is the top–level directory where the Meridian Ada compiler was installed.

Note: Always make a backup of the old **newlib.bat** file before modifying it.

2.4 Verifying Correct Installation

Below the top–level directory given to **install**, the directory **dosenv** and **test** should have been created by the installation procedure for the *DOS Environment Library*. Note that if you installed the *DOS Environment Library* in the same top–level directory in which you originally installed the compiler (e.g. **c:\ada**) the **test** directory already existed.

There should be a number of files present in the **dosenv** directory: ***.lib**, ***.aar**, and ***.int**.

In the **test** directory these two source files should be present:

boxdemo.ada — Draws boxes on the screen. In addition to demonstrating several of the DOS Environment packages, it also demonstrates Meridian Ada tasking.
NOTE: Because this program uses the screen management packages, it requires a 100% IBM–compatible BIOS.

envdisp.ada — Displays current environment settings. This prints the current directory, the names of files in the current directory, environment variable values, and other environment–specific information.

To compile and run these demonstration programs, follow these steps:

1. Go to the **test** directory and enter a link in the local library to the *DOS Environment Library* library, as in this example:

   ```
   c:
   cd ada\test
   newlib
   lnlib c:\ada\dosenv\ada.lib
   rem --If newlib has been modified to do this
   rem --already, you can skip the lnlib command.
   ```

   This example assumes that the *DOS Environment Library* was installed using **c:\ada** as the top–level directory. Use whatever path is appropriate for your installation.

2. Compile and link the first sample program:

   ```
   ada boxdemo.ada
   bamp boxdemo
   ```

3. Assuming that all went well, run the sample program as:

   ```
   boxdemo
   ```

   This should clear the display and draw some boxes. If it cannot do this with the current display mode, an error message is printed. Regardless of precisely what is displayed, something should happen.

4. Compile and link the second sample program:

   ```
   ada envdisp.ada
   bamp envdisp
   ```

5. Assuming that all went well, run the second sample program as:

   ```
   envdisp
   ```

   This should print various information about the current environment, as detailed in the program description above. Regardless of precisely what is displayed, something should happen.

If these programs work correctly (within their operational parameters), then the *DOS Environment Library* was probably loaded correctly and is ready for further use in other programs.

Chapter 3 Package Absolute_Disk

Package **absolute_disk** provides procedures to read and write on disk devices without regard to the file system structure imposed by PC–DOS.

SPECIFICATION

```
with disk_types,
     system;
package absolute_disk is
   type status is (
      ok,                  bad_command,
      address_not_found,   write_protected,
      sector_not_found,    dma_failure,
      bad_crc,             controller_failed,
      seek_failed,         time_out,
      unknown_error
    );
   procedure  read  (
      from_drive               : disk_types.drive_id;
      number_of_sectors        : natural;
      starting_logical_sector  : natural;
      transfer_area            : system.address;
      error                    : out status
    );
   procedure write (
      from_drive               : disk_types.drive_id;
      number_of_sectors        : natural;
      starting_logical_sector  : natural;
      transfer_area            : system.address;
      error                    : out status
    );
end absolute_disk;
```

3.1 Procedure Read

The procedure **read** gets data from specific logical disk sectors, ignoring the file system structure of the disk. The selected disk seeks to the *starting_logical_sector*, reads the *number_of_sectors* specified into the memory location starting at the *transfer_area*, and returns an *error* status. If no error occurs, **status' (ok)** is returned. Any other status value indicates some kind of error.

```
procedure read (
   from_drive               : disk_types.drive_id;
   number_of_sectors        : natural;
   starting_logical_sector  : natural;
   transfer_area            : system.address;
   error                    : out status
 );
```

The number of bytes read per sector is typically 512, but this number may vary according to the disk device type.

Logical sectors are numbered sequentially from track 0, head 0, sector 1, and continue until the last sector, the last sector number being a characteristic of a specific type of disk device. As logical sector numbers increase, the internal sector, head, and track numbers increment (in that order).

The **read** service works with logical drives as opposed to true physical drives. An attempt to read from a logical drive that has no corresponding physical drive or disk area (i.e is not installed) has unpredictable results.

This call corresponds to Interrupt 16#25#.

3.2 Procedure Write

The procedure **write** puts data to specific logical disk sectors, ignoring the file system structure of the disk. The selected disk seeks to the *starting_logical_sector*, writes the *number_of_sectors* specified from the memory location starting at the *transfer_area*, and returns an *error* status. If no error occurs, **status' (ok)** is returned. Any other status value indicates some kind of error.

```
procedure write (
   from_drive              : disk_types.drive_id;
   number_of_sectors       : natural;
   starting_logical_sector : natural;
   transfer_area           : system.address;
   error                   : out status
  );
```

The number of bytes written per sector is typically 512, but this number may vary according to the disk device type.

Logical sectors are numbered sequentially from track 0, head 0, sector 1, and continue until the last sector, the last sector number being a characteristic of a specific type of disk device. As logical sector numbers increase, the internal sector, head, and track numbers increment (in that order).

The **write** service works with logical drives as opposed to true physical drives. An attempt to write to a logical drive that has no corresponding physical drive or disk area (i.e is not installed) has unpredictable results.

** WARNING ** Writing sectors directly to a disk in this way is likely to destroy any previous disk file structure.

This call corresponds to Interrupt 16#26#.

Chapter 4 Package Box

The package **box** provides procedures for drawing boxes on the text screen either using the box–drawing characters (tops, bottoms, sides, and corners) or using a single selected character.

The routines in package **box** call many DOS functions; there is no correspondence to a single DOS function as with most of the other subprograms in the *Meridian Ada DOS Environment Library*.

SPECIFICATION

```
with common_display_types;
use  common_display_types;

package box is

   type  part  is  (
      north, northeast, east, southeast,
      south, southwest, west, northwest
     );

   type user_definition is array (part) of extended_ascii;

   type simple_kind is (
      single_sided, double_sided,
      single_top,   double_top
     );

procedure draw (
   upper_left_row      : row_range;
   upper_left_column   : column_range;
   lower_right_row     : row_range;
   lower_right_column  : column_range;
   kind                : simple_kind  := single_sided;
   attribute           : display_attribute := (
      foreground => white,
      background => black,
      blink      => false
     );
   page                : display_page      := 0
  );
```

```
procedure draw (
   upper_left_row     : row_range;
   upper_left_column  : column_range;
   lower_right_row    : row_range;
   lower_right_column : column_range;
   box_definition     : user_definition;
   attribute          : display_attribute := (
      foreground => white,
      background => black,
      blink      => false
     );
   page               : display_page      := 0
  );
end box;
```

4.1 Type Part

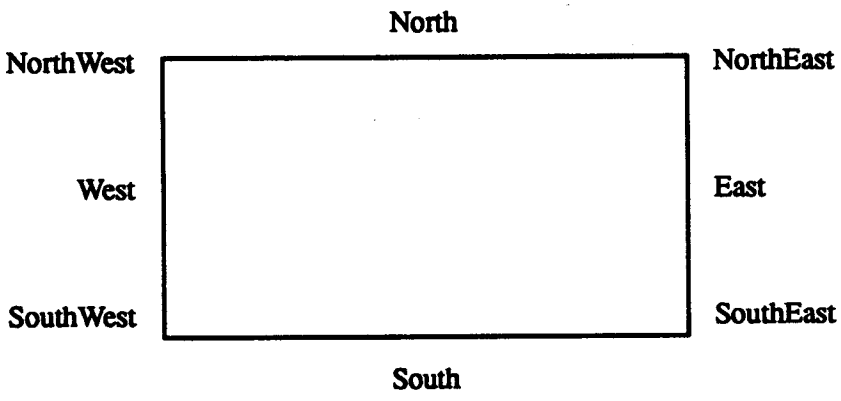
Type **part** is an enumeration used to describe the parts of a box.

```
type part is (
     north, northeast, east, southeast,
     south, southwest, west, northwest
  );
```

The elements are compass points and have these correspondences:

North	The top of a box
NorthEast	The upper right corner of a box
East	The right side of a box
SouthEast	The lower right corner of a box
South	The bottom of a box
SouthWest	The lower left corner of a box
West	The left side of a box
NorthWest	The upper left corner of a box

These are illustrated as:



4.2 Type User_Definition

Type **user_definition** is used to describe what character is used for each part of a box.

```
type user_definition is array (part) of extended ascii;
```

Some particularly useful characters for drawing boxes are in the range 179 .. 218. If no table showing these characters is available, then the following program will print them out:

```
with video;
procedure printchars is
begin
   for char in 179 .. 218 loop
      video.write_tty (char); -- write to page 0
      video.write_tty (' ');  -- separate with space
   end loop;
end printchars;
```

4.3 Type Simple_Kind

Type **simple_kind** is used to select a pre–defined box style.

```
type simple_kind is (
   single_sided, double_sided, .
   single_top,   double_top
  );
```

The default box–drawing characters are the IBM PC characters used to draw connecting horizontal and vertical lines and corners, with either single rules or double rules.

The box styles corresponding to each **simple_kind** are:

single_sided	single–rule lines all around
double_sided	double–rule lines all around
single_top	single–rule lines on top and bottom; double rule lines on the sides
double_top	double–rule lines on top and bottom; single rule lines on the sides

4.4 Procedure Draw

There are two overloaded versions of procedure **draw**, the disambiguating parameter being **kind** or **box_definition**. The first version draws a few kinds of boxes using default box–drawing characters, the second version uses selected characters for each part of a box.

4.4.1 Using Default Boxes

The first version of procedure **draw** displays one of several pre–defined box types using default box–drawing characters.

```
procedure draw (
   upper_left_row      : row_range;
   upper_left_column   : column_range;
   lower_right_row     : row_range;
   lower_right_column  : column_range;
   kind                : simple_kind := single_sided;
   attribute           : display_attribute := (
      foreground => white,
      background => black,
      blink      => false
     );
   page                : display_page := 0
  );
```

Given the upper left and lower right coordinates, a box is drawn on the specified display page with the specified style and attributes.

This call is valid for text modes only.

Invalid coordinates or attempting to exceed the current mode's display size has unpredictable results.

An example for drawing a single–sided box follows.

```
-- draw a single-sided box at (row/col)
--
--      5,5-----------5,20
--        |             |
--        |             |
--     10,5----------10,20
--
-- in page zero, with White/Black blinking attributes.
--
with  common_display_types,
      box;
use   common_display_types;
procedure box_test1 is
   attr  : display_attribute;
begin
   attr.blink := true;
   box.draw (
      upper_left_row     => 5,
      upper_left_column  => 5,
      lower_right_row    => 10,
      lower_right_column => 20,
      attribute          => attr
     );
end;
```

4.4.2 Using a Selected Character

The second version of procedure **draw** uses a selected character to draw each box part.

```
procedure draw (
   upper_left_row      : row_range;
   upper_left_column   : column_range;
   lower_right_row     : row_range;
   lower_right_column  : column_range;
   box_definition      : user_definition;
   attribute           : display_attribute := (
      foreground => white,
      background => black,
      blink      => false
     );
   page                : display_page := 0
  );
```

Given the upper left and lower right coordinates, a box is drawn with the specified character and attributes in the specified display page.

This call is valid for text modes only.

Invalid coordinates or attempting to exceed the current mode's display size has unpredictable results.

An example of procedure **draw** using a selected character follows.

```
-- draw a box with the Happy Face character at (row/col)
--
--       5,5----------5,30
--         |             |
--         |             |
--       7,5----------7,30
--
-- in page zero, with Cyan/Black attributes.
--
with  common_display_types,
      box;
use   common_display_types;
procedure box_test2 is
   attr  : display_attribute;
   my_box   : box.user_definition := (others => 1);
      -- 1 = Happy Face
begin
   attr.foreground := cyan;
   box.draw (
      upper_left_row      => 5,
      upper_left_column   => 5,
      lower_right_row     => 7,
      lower_right_column  => 30,
      box_definition      => my_box,
      attribute           => attr
     );
end;
```

Note that a different box–drawing character from the IBM PC character set may be used for each part.

Chapter 5 Package Common_Display_Types

The package **common_display_types** contains type declarations for the various packages that handle display operations: **box**, **cursor**, **tty**, and **video**.

SPECIFICATION

```
package common_display_types is
   subtype byte is integer range 0 .. 255;
   subtype extended_ascii is integer range 0 .. 255;
   subtype column_range     is integer range 0 .. 79;
   subtype row_range        is integer range 0 .. 24;
   subtype cursor_size is integer range 0 .. 13;
   subtype display_page is integer range 0 .. 7;
   type color is (
      black,        blue,           green,
      cyan,         red,            magenta,
      brown,        white,          grey,
      light_blue,   light_green,    light_cyan,
      light_red,    light_magenta,  yellow,
      bright_white
     );
   subtype background_color is color range black .. white;
   type display_attribute is
      record
         foreground : color            := white;
         background : background_color := black;
         blink      : boolean          := false;
      end record;
   --
   -- underline valid for monochrome mode only.
   --
   underline : constant display_attribute := (
      foreground => blue,
      background => black,
      blink      => false
     );
   subtype color_palette is integer range 0 .. 1;
   type graphic_color is (
      background,   color1,
      color2,       color3
     );
end common_display_types;
```

Chapter 6 Package Cursor

Package **cursor** provides operations to move a text mode cursor, to get the current cursor coordinates, and to alter the visibility or form of the cursor.

The Color/Graphics Adapter has eight display pages (0 .. 7) in 40 column by 25 line text modes and four display pages in the 80 column by 25 line text modes. The Monochrome adapter has only one display page. Specifying an invalid display page (i.e. a display page that is not defined for the current text display mode) has unpredictable results.

SPECIFICATION

```
with     common_display_types;
use      common_display_types;
package cursor is

   procedure set_size (
      start_line : cursor_size;
      end_line   : cursor_size
    );

   procedure inhibit;

   procedure move (
      row      : row_range;
      column   : column_range;
      page     : display_page := 0
     );

   procedure get_position (
      row      : out row_range;
      column   : out column_range;
      page     : display_page  :=  0
     );

   procedure  up    (page :  display_page  := 0);
   procedure  down  (page :  display_page  := 0);
   procedure  left  (page :  display_page  := 0);
   procedure  right (page :  display_page  := 0);

end cursor;
```

6.1 Procedure Set_Size

Procedure **set_size** sets the *start_line* and *end_line* of the blinking cursor.

```
procedure set_size (
   start_line :  cursor_size;
   end_line   :  cursor_size
  );
```

This call is valid only in Text modes.

The ROM BIOS default values for *start_line* and *end_line* are:

text display mode	Start_Line	End_Line
Text80_BW_Ma (Mode 7)	11	12
Text40_BW (Mode 0)	6	7
Text40_CO (Mode 1)	6	7
Text80_BW (Mode 2)	6	7
Text80_CO (Mode 3)	6	7

A particular text display mode is selectable via the procedure **video.set**. Refer to package **video** for a description of the various text modes.

This call corresponds to Interrupt 16#10#, function 16#01#.

6.2 Procedure Inhibit

Procedure **inhibit** inhibits display of the cursor (i.e. makes it invisible).

```
procedure inhibit;
```

This call is valid only in Text modes.

This call corresponds to Interrupt 16#10#, function 16#01#.

6.3 Procedure Move

Procedure **move** moves the cursor to the specified *row* and *column* within the specified display *page*.

```
procedure move (
   row      : row_range;
   column   : column_range;
   page     : display_page := 0
  );
```

Moving the cursor to a location that is invalid for the current display mode has unpredictable results.

The coordinate scheme is rectangular:

(0,0) Coordinate (Row 0, Column 0) is the upper left corner of the display.

(24,79) Coordinate (Row 24, Column 79) is the lower right corner of the display in 80–column text modes.

This call corresponds to Interrupt 16#10#, function 16#02#.

An example follows:

```
-- Move the cursor to row 10, column 20 in display page
-- 0 (default page).
--
cursor.move (row => 10, column => 20);
```

6.4 Procedure Get_Position

Procedure **get_position** returns the current cursor position for the specified display page.

```
procedure get_position (
   row      : out row_range;
   column   : out column_range;
   page     : display_page := 0
  );
```

This call corresponds to Interrupt 16#10#, function 16#03#.

An example follows.

```
-- Get the current cursor position of display page 1
--
cursor.get_position (
   row      => page1_row,
   column   => page1_col,
   page     => 1
  );
```

6.5 Procedure Up

Procedure **up** moves the cursor up one location in the specified display page.

```
procedure up (page : display_page := 0);
```

Attempts to move the cursor up past the top of the screen are ignored.

This call corresponds to Interrupt 16#10#, function 16#02#.

An example follows.

```
-- Move the display page zero cursor up one location.
--
   cursor.up;
```

6.6 Procedure Down

Procedure **down** moves the cursor down one location in the specified display page.

```
procedure down (page : display_page := 0);
```

Attempts to move the cursor down past the bottom of the screen are ignored.

This call corresponds to Interrupt 16#10#, function 16#02#.

An example follows.

```
-- Move the display page one cursor down one location.
--
   cursor.down (page => 1);
```

6.7 Procedure Left

Procedure **left** moves the cursor left one location in the specified display page.

```
procedure left (page : display page := 0);
```

Attempts to move the cursor left past the edge of the screen are ignored.

This call corresponds to Interrupt 16#10#, function 16#02#.

6.8 Procedure Right

Procedure **right** moves the cursor right one location in the specified display page.

```
procedure right (page : display page := 0);
```

Attempts to move the cursor past the right edge of the screen are ignored.

This call corresponds to Interrupt 16#10#, function 16#02#.

Chapter 7 Package Directory

Package **directory** provides subprograms that operate on directories. A program using this package can create or delete directories and can change the current directory of the program.

SPECIFICATION

```
with errors,
     disk_types;
package directory is
   subtype pathname is string (1 .. 65);
   function make (name : in string)
      return errors.extended_errors;
   function remove (name : in string)
      return errors.extended_errors;
   function change_to (name : in string)
      return errors.extended_errors;
   procedure current_name (
      for_drive  : disk_types.drive_designator;
      name       : out pathname;
      last       : out natural;
      error      : out errors.extended_errors
     );
end directory;
```

7.1 Function Make

Function **make** creates a directory.

```
function make (name : in string) return
   errors.extended.errors;
```

This call corresponds to Interrupt 16#21#, function 16#39#.

Possible errors are:

- **errors.path_not_found**
- **errors.access_denied**

Examples which use function **make** follow.

```
-- Create the directory TEST.DIR in the root directory
-- of the 'C' Drive.
--
with errors,
     directory;
procedure dir_test1 is
   error : errors.extended_errors;
begin
   error := directory.make (name => "C:\TEST.DIR");
end;
```

Another example follows.

```
-- Create the directory TEST.DIR in the current directory
-- on the default disk drive.
--
with errors,
     directory;
procedure dir_test2 is
   error : errors.extended_errors;
begin
   error := directory.make (name => "TEST.DIR");
end;
```

7.2 Function Remove

Function **remove** removes a directory.

```
function remove (name : in string) return
   errors.extended_errors;
```

This call corresponds to Interrupt 16#21#, function 16#3A#.

Possible errors are:

- **errors.path_not_found**
- **errors.access_denied**
- **errors.remove_current_directory**

An example which uses function **remove** follows.

```
-- Remove TEST.DIR from the current drive and directory.
with directory,
     errors;
procedure remove_test is
   error : errors.extended_errors;
begin
   error := directory.remove (name => "TEST.DIR");
end;
```

7.3 Function Change_To

Changes the current directory for the program.

```
function change_to (name : in string)
   return errors.extended_errors;
```

This call corresponds to Interrupt 16#21#, function 16#3B#.

A possible error follows:

- **errors.path_not_found**

An example of using package **directory** with the function **change_to** follows.

```
-- Change the current working directory to the parent directory
--
with directory,
     errors;
procedure change_test is
   error : errors.extended_errors;
begin
   error := directory.change_to (name => "..");
end;
```

7.4 Procedure Current_Name

Procedure **current_name** obtains the pathname of the current working directory for the specified drive.

```
procedure current_name (
   for_drive  : disk_types.drive_designator;
   name       : out pathname;
   last       : out natural;
   error      : out errors.extended_errors
  );
```

The returned pathname does not include the drive identifier.

The returned parameter *last* denotes the index of the last valid character position in *name*.

This call corresponds to Interrupt 16#21#, function 16#47#.

A possible error follows:

- **errors.invalid_drive**

An example of using package **directory** with procedure **current_name** follows.

```
-- Get and display the current directory for the 'A' drive.
--
with directory,
     disk_types,
     errors,
     text_io;
use  disk_types,
     directory;
procedure display_dir is
   current_a_dir      : pathname;
   last_valid_char    : natural;
   current_name_error : errors.extended_errors;
begin
   current_name (
   for_drive  => a,
   name       => current_a_dir,
   last       => last_valid_char,
   error      => current_name_error
  );
   text_io.put_line (current_a_dir (1 .. last_valid_char));
end;
```

Chapter 8 Package Disk

Package **disk** provides operations on disk devices.

SPECIFICATION

```
with disk_types;
package disk is
   procedure reset;
   function set_default (to_drive : disk_types.drive_id)
      return disk_types.logical_drive_count;
   function get_default return disk_types.drive_id;
   -- Get File Allocation Information
   --
   procedure get_allocation_info (
      for_drive         : disk_types.drive_designator;
      sectors           : out natural;
      bytes_per_sector  : out natural;
      clusters          : out natural;
      fat_id_byte       : out disk_types.identification;
      error             : out boolean
     );
   procedure set_verification (is_on : boolean);
   function verification_is_on return boolean;
   procedure get_free_space (
      for_drive           : disk_types.drive_designator;
      sectors_per_cluster : out natural;
      available_clusters  : out natural;
      bytes_per_sector    : out natural;
      total_clusters      : out natural;
      error               : out boolean
     );
   procedure get_free_space (
      for_drive     : disk_types.drive_designator;
      free_bytes    : out long_integer;
      total_space   : out long_integer;
      error         : out boolean
     );
end disk;
```

8.1 Procedure Reset

Procedure **reset** flushes all file buffers to disk without closing any files.

```
procedure reset;
```

This call corresponds to Interrupt 16#21#, function 16#0D#.

8.2 Function Set_Default

Function **set_default** sets the current default drive and returns the number of logical drives installed.

```
function set default (to drive : disk_types.drive id)
   return disk_types.logical_drive_count;
```

Logical drive means any block device: ram disk, floppy disk, or hard disk.

This call corresponds to Interrupt 16#21#, function 16#0E#.

An example of using package **disk** with function **set_default** follows.

```
-- set the current drive to the 'A' drive.
--
with disk, disk_types;
use  disk, disk_types;
procedure disk_test1 is
   count : disk_types.logical_drive_count;
begin
   count := set_default (to_drive => a);
end;
```

8.3 Function Get_Default

Function **get_default** returns the drive ID of the default disk drive.

```
function get_default return disk_types.drive_id;
```

This call corresponds to Interrupt 16#21#, function 16#19#.

8.4 Procedure Get_Allocation_Info

Procedure **get_allocation_info** obtains information about the specified disk drive.

```
procedure get_allocation_info (
   for_drive          : disk_types.drive_designator;
   sectors            : out natural;
   bytes_per_sector   : out natural;
   clusters           : out natural;
   fat_id_byte        : out disk_types.identification;
   error              : out boolean
  );
```

This call corresponds to Interrupt 16#21#, function 16#1C#.

8.5 Procedure Set_Verification

Procedure **set_verification** turns on or off the read–after– write verification of all data written to disk.

```
procedure set_verification (is_on : boolean);
```

This call corresponds to Interrupt 16#21#, function 16#2E#.

An example of using package **disk** with procedure **set_verification** follows.

```
-- set verification to on
--
disk.set_verification (is_on => true);
```

8.6 Function Verification_Is_On

Function **verification_is_on** determines whether the read– after–write verification flag is on or off.

```
function verification_is_on return boolean;
```

This call corresponds to Interrupt 16#21#, function 16#54#.

8.7 Procedure Get_Free_Space

Procedure **get_free_space** obtains selected information for the specified disk drive.

```
procedure get_free_space (
   for_drive           : disk_types.drive_designator;
   sectors_per_cluster : out natural;
   available_clusters  : out natural;
   bytes_per_sector    : out natural;
   total_clusters      : out natural;
   error               : out boolean
  );
```

An error may be caused by an invalid drive designator.

This call corresponds to Interrupt 16#21#, function 16#36#.

8.8 Procedure Get_Free_Space

Procedure **get_free_space** obtains the amount of total space and amount of free space for the specified disk drive.

```
procedure get_free_space (
   for_drive     : disk_types.drive_designator;
   free_bytes    : out long_integer;
   total_space   : out long_integer;
   error         : out boolean
  );
```

An error may be caused by an invalid drive designator.

This call corresponds to Interrupt 16#21#, function 16#36#.

An example of using package **disk** with procedure **get_free_space** follows.

```
-- Get and display the used space of the 'A' disk drive.
--
with disk,
     text_io;
use  disk;
procedure display_used_on_a is
```

```
   total_free     : long_integer;
   disk_size      : long_integer;
   space_error    : boolean;
begin
   get_free_space (
      for_drive      => a;
      free_byte      => total_free;
      total_space    => disk_size;
      error          => space_error
    );
   if not space_error then
      text_io.put_line (
         "Total Used is " &
         long_integer'image (disk_size - total_free));
   end if;
end;
```

Chapter 9 Package Disk_Types

This package contains type declarations used by the disk and directory operation packages **absolute_disk**, **directory**, and **disk**.

SPECIFICATION

```
package disk_types is

   type drive_designator is (
       current_drive, a, b, c, d, e,
       f, g, h, i, j, k, l, m, n, o,
       p, q, r, s, t, u, v, w, x, y,
       z
     );

   subtype drive_id is drive_designator range a .. z;

   subtype logical_drive_count is integer range 0 .. 255;
   subtype identification      is integer range 0 .. 255;

end disk_types;
```

Chapter 10 Package Equipment

Package **equipment** provides a function that enumerates the system–recognized devices and facilities present on the machine.

SPECIFICATION

```
package equipment is
   type mode is (not_used, color40, color80, mono80);
   type equipment_list is
      record
       printers            : integer;
       serial_printer      : boolean;
       game_adapter        : boolean;
       rs232_ports         : integer;
       dma_present         : boolean;
       diskette_drives     : integer;
       initial_video_mode  : mode;
       system_ram          : integer;
       math_coprocessor    : boolean;
       diskette_present    : boolean;
      end record;
   function list return equipment_list;
end equipment;
```

10.1 Function List

Function **list** returns the list of currently installed equipment.

```
function list return equipment_list;
```

The *equipment_list* information corresponds to the word at address 0000:0410H.

This call corresponds to Interrupt 16#11#.

An example of using package **equipment** with function **list** follows.

```
-- determine if a game adapter is present.
with equipment;  use  equipment;
procedure equipment_test1 is
   my_equipment : equipment_list;
begin
   my_equipment := equipment.list;
   if my_equipment.game_adapter then
      null;
   end if;
end;
```

Chapter 11 Package Errors

Package **errors** provides PC–DOS status declarations and for version 3.0 or later of PC–DOS, a procedure to get extended status information.

SPECIFICATION

```
package errors is

   type extended_errors is (
        ok,
        invalid_function,
        file_not_found,
        path_not_found,
        no_handle_available,
        access_denied,
        invalid_handle,
        memory_blocks_destroyed,
        insufficient_memory,
        invalid_memory_block,
        invalid_environment,
        invalid_format,
        invalid_file_access,
        invalid_data,
        reserved14,
        invalid_drive,
        remove_current_directory,
        not_same_device,
        no_more_files,
        disk_write_protected,
        unknown_unit,
        drive_not_ready,
        command_not_defined,
        disk_data_error,
        bad_structure_length,
        seek_error,
        unknown_media_type,
        sector_not_found,
        printer_outof_paper,
        write_error,
        read_error,
        general_failure,
        file_sharing_violation,
        file_locking_violation,
        invalid_disk_change,
        no_fcb_available,
```

```
        file_already_exists,
        reserved81,
        cannot_make,
        interrupt_failure,
        unknown_error
      );

    type class_code is (
        out_of_resource,          temporary_situation,
        authorization,            internal_dos_error,
        hardware_failure,         system_sw_error,
        application_error,        not_found,
        bad_format,               item_locked,
        media_error,              already_exists,
        unknown
      );

    type action_code is (
        try_again,                try_again_later,
        user_to_fix,              shut_down_program,
        shut_down_immediate,      ignore_error,
        retry_after_fix,          unknown
      );

    type locus_code is (
        unknown,                  block_device,
        network_related,          serial_device,
        memory_related
      );

    -- DOS 3.0 procedure
    procedure get_extended_info (
        error       : out extended_errors;
        class       : out class_code;
        action      : out action_code;
        locus       : out locus_code
      );

end errors;
```

11.1 Procedure Get_Extended_Info

Procedure `get_extended_info` obtains detailed information after a previously unsuccessful system call. The information may suggest action that the program should take.

```
procedure get_extended_info (
    error           : out extended_errors;
    class           : out class_code;
    action          : out action_code;
    locus           : out locus_code
  );
```

This function requires DOS 3.0 or higher. Execution of this function with a lower revision causes the exception `revision.incorrect_dos_version` to be raised.

This call corresponds to Interrupt 16#21#, function 16#59#.

An example follows.

```
with errors,
     directory;
use  errors;
procedure errors_test1 is

  make_error : extended_errors;
  error  : extended_errors;
  class  : class_code;
  action : action_code;
  locus  : locus_code;
begin
   make_error := directory.make (name => "A:\Junk");
   if make_error /= ok then
      get_extended_info (error, class, action, locus);
   end if;
end;
```

Chapter 12 Package File_IO

Package **file_io** provides file input–output, file attribute manipulation, and wildcard search operations. Note that most of the essential input–output facilities provided in this package are available in a largely system–independent standard package, **text_io**, which is distributed with the Meridian Ada compiler.

Be aware that the Ada predefined type **character** supports only seven–bit ASCII characters. If it is necessary to manipulate all eight bits of a byte, then pragma **suppress** should be used when using objects of type **character** (to avoid the exception **constraint_error**), or a different type altogether (e.g. **common_display_types.extended_ascii**) should be used.

SPECIFICATION

```
with errors,
     time,
     system;

package file_io is

   type file_handle is new integer;

   -- Standard I/O Handles
   stdin   : constant file_handle := 0;  -- CON
   stdout  : constant file_handle := 1;  -- CON
   stderr  : constant file_handle := 2;  -- CON
   stdaux  : constant file_handle := 3;  -- AUX
   stdlist : constant file_handle := 4;  -- PRN

   type file_attribute_types is (
       read_only,    hidden,       system,
       volume_label, subdirectory, archive
     );

   type file_attributes is array (file_attribute_types)
       of boolean;

   --
   -- Access Code Values
   --
   type access_mode is (
       read_only,
       write_only,
       read_write
     );
   type sharing_mode is (
       compatibility, deny_read_write, deny_write,
       deny_read,     deny_none
     );
```

```
type access_code is
   record
    inheritance_flag : boolean      := true;
    -- DOS 3.0 only
    sharing          : sharing_mode := compatibility;
    -- DOS 3.0 only
    reserved_bit     : boolean      := false;
    -- DOS 3.0 only

    file_access_mode : access_mode  := read_write;
   end record;

type standard_io is (input, output);

type method_code is (
    beginning,
    current_location,
    current_end
  );

subtype timestamp_data is time.packet;

type transfer_data is limited private;

type file_data is
   record
    name       : string (1..12);
    attributes : file_attributes;
    timestamp  : timestamp_data;
    file_size  : long_integer; -- in bytes
   end record;

procedure create (
    name       : string;
    handle     : out file_handle;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                   (others => false)
  );

-- DOS 3.0
procedure create_temporary (
    pathname   : string;
    handle     : out file_handle;
    temp_file  : out string;
    last       : out natural;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                   (others => false)
   );
```

```
-- DOS 3.0
procedure create_temporary (
    handle      : out file_handle;
    temp_file   : out string;
    last        : out natural;
    error       : out errors.extended_errors;
    attributes  : file_attributes :=
                    (others => false)
  );
-- DOS 3.0
procedure create_new (
    name        : string;
    handle      : out file_handle;
    error       : out errors.extended_errors;
    attributes  : file_attributes :=
                    (others => false)
  );
procedure open (
    name   : string;
    handle : out file_handle;
    error  : out errors.extended_errors;
    code   : access_code := (
        inheritance_flag => true,
        sharing          => compatibility,
        reserved_bit     => false,
        file_access_mode => read_write
      )
  );
function close (handle : file_handle)
    return errors.extended_errors;
function cooked (for_standard : standard_io)
    return errors.extended_errors;
function raw    (for_standard : standard_io)
    return errors.extended_errors;
procedure read (
    handle          : file_handle;
    bytes_to_read   : natural;
    buffer_address  : system.address;
    bytes_read      : out natural;
    error           : out errors.extended_errors
  );
procedure write (
    handle          : file_handle;
    bytes_to_write  : natural;
    buffer_address  : system.address;
    bytes_written   : out natural;
    error           : out errors.extended_errors
  );
```

```
function delete (name : string) return
    errors.extended_errors;

function rename (
    old_name : string;
    new_name : string
  ) return errors.extended_errors;

procedure move_file_pointer (
    handle         : file_handle;
    offset_method  : method_code;
    offset_value   : long_integer;
    new_pointer    : out long_integer;
    error          : out errors.extended_errors
  );

--
-- Get and Set File Attributes
--
procedure get_attributes (
    name       : string;
    attributes : out file_attributes;
    error      : out errors.extended_errors
  );

function set_attributes (
    name       : string;
    attributes : file_attributes
  ) return errors.extended_errors;

--
-- Get and Set File Date and Time
--
function set_file_time (
    handle : file_handle;
    time   : timestamp_data
  ) return errors.extended_errors;

procedure get_file_time (
    handle : file_handle;
    time   : out timestamp_data;
    error  : out errors.extended_errors
  );

procedure dup_filehandle (
    handle     : file_handle;
    new_handle : out file_handle;
    error      : out errors.extended_errors
  );

procedure cdup_filehandle (
    handle1    : file_handle;
    handle2    : file_handle;
    error      : out errors.extended_errors
  );
```

```
    --
    -- File Search functions
    --

    procedure find_first (
        name_template     : string;
        transfer_area     : out transfer_data;
        file_info         : out file_data;
        error             : out errors.extended_errors;
        search_attributes: file_attributes :=
                              (others => false)
      );

    procedure find_next (
        transfer_area : in out transfer_data;
        file_info     : out file_data;
        error         : out errors.extended_errors
      );

private

   type transfer_data is array (0 .. 21) of integer;

end file_io;
```

12.1 Procedure Create

Procedure **create** creates a file in the current or specified directory, and opens it for program use.

```
procedure create (
    name       : string;
    handle     : out file_handle;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                    (others => false)
  );
```

If the file already exists, then it is truncated to zero. The file is opened for read/write access.

File attributes **volume** and **subdirectory** are ignored by this call.

This call corresponds to Interrupt 16#21#, function 16#3C#.

Possible errors are:

- **errors.path_not_found**
- **errors.no_handle_available**
- **errors.access_denied**

An example of using package **file_io** with procedure **create** follows.

```
-- create "Test.Out" with normal attributes in the
-- current directory on the default drive.
--
with file_io, errors, text_io;
use  file_io, errors;
procedure fio_test1 is
```

```
  test_handle   : file_handle;
  create_error : errors.extended_errors;
begin
  create ("Test.Out", test_handle, create_error);
  if create_error = ok then
     -- use test_handle for any read or write requests.
     null;
  else
     text_io.put_line (
       "Error on Create !! : " &
       errors.extended_errors'image (create_error)
       );
  end if;
end;
```

12.2 Procedure Create_Temporary

There are two overloaded versions of procedure **create_temporary** disambiguated by the presence or absence of the parameter *pathname*. The first version creates a temporary file in the directory specified by the *pathname*; the second version creates a temporary file in the current directory.

12.2.1 Using *Pathname*

With a *pathname* parameter, procedure **create_temporary** creates a temporary file in the directory specified by *pathname*, and opens the file.

```
-- DOS 3.0
procedure create_temporary (
    pathname   : string;
    handle     : out file_handle;
    temp_file  : out string;
    last       : out natural;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                   (others => false)
  );
```

The parameter *temp_file* is set to the path and name of the temporary file. Parameter *last* is the index position of the last valid character of the *temp_file* string.

The temporary file is not automatically deleted at program completion.

The file is opened for read/write access.

File attributes **volume** and **subdirectory** are ignored by this call.

This function requires DOS 3.0 or higher. Execution of this function with a lower revision causes the exception **revision.incorrect_dos_version** to be raised.

This call corresponds to Interrupt 16#21#, function 16#5A#.

Possible errors are:

- **errors.path_not_found**
- **errors.access_denied**

An example of using package **file_io** with procedure **create_temporary** follows.

```
-- Create a temporary file in the directory specified
-- by "A:\TEMP.DIR" (a:\temp.dir must already exist)
--
with file_io, errors;
use  file_io;
procedure fio_test2 is

  temp_handle   : file_handle;
  temp_filename : string;
  last_char     : natural;
  create_error  : errors.extended_errors;
begin
  create_temporary (
      pathname   => "A:\TEMP.DIR",
      handle     => temp_handle,
      temp_file  => temp_filename,
      last       => last_char,
      error      => create_error
    );
end;
```

12.2.2 Using the Current Directory

Without a *pathname* parameter, procedure **create_temporary** creates a temporary file in the current directory on the default disk drive and opens the file.

```
-- DOS 3.0
procedure create_temporary (
    handle     : out file_handle;
    temp_file  : out string;
    last       : out natural;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                   (others => false)
  );
```

The parameter *temp_file* is set to the name of the temporary file. Parameter *last* is the index position of the last valid character of the *temp_file* string.

The temporary file is not automatically deleted at program completion.

The file is opened for read/write access.

File attributes **volume** and **subdirectory** are ignored by this call.

This function requires DOS 3.0 or higher. Execution of this function with a lower revision causes the exception **revision.incorrect_dos_version** to be raised.

This call corresponds to Interrupt 16#21#, function 16#5A#.

Possible errors are:

- **errors.path_not_found**
- **errors.access_denied**

12.3 Procedure Create_New

Procedure **create_new** creates a file in the current or specified directory, and opens the file. If the file already exists, then the **file_already_exists** error is returned.

```
-- DOS 3.0
procedure create_new (
    name       : string;
    handle     : out file_handle;
    error      : out errors.extended_errors;
    attributes : file_attributes :=
                   (others => false)
);
```

The file is opened for read/write access.

File attributes **volume** and **subdirectory** are ignored by this call.

This call corresponds to Interrupt 16#21#, function 16#5B#.

This function requires DOS 3.0 or higher. Execution of this function with a lower revision causes the exception **revision.incorrect_dos_version** to be raised.

Possible errors are:

- **errors.path_not_found**
- **errors.no_handle_available**
- **errors.access_denied**
- **errors.file_already_exists**

An example of using package **file_io** with procedure **create_new** follows.

```
-- Create "TEST" as a Read_Only file in the current
-- directory.
--
with file_io, errors,
use  file_io;
procedure fio_test3 is
   new_handle   : file_handle;
   create_error : errors.extended_errors;
   file_attr    : file_attributes :=
                    (others => false);
begin
   file_attr (read_only) := true;
   create_new (
       name       => "Test",
       handle     => new_handle,
       error      => create_error,
       attributes => file_attr
     );
end;
```

12.4 Procedure Open

Procedure **open** opens an existing file.

```
procedure open (
    name   : string;
    handle : out file_handle;
    error  : out errors.extended_errors;
    code   : access_code := (
       inheritance_flag => true,
       sharing          => compatibility,
       reserved_bit     => false,
       file_access_mode => read_write
      )
);
```

For DOS revision 2, the *file_access_mode* is the only parameter actually used; the parameters *inheritance_flag* and *sharing* are ignored. All the parameters are used in revision 3.0 and higher.

Normal files and files with the **system** and **hidden** attributes can be opened with this call.

This call corresponds to Interrupt 16#21#, function 16#3D#.

Possible errors are:

- **errors.invalid_function**
- **errors.file_not_found**
- **errors.path_not_found**
- **errors.no_handle_available**
- **errors.access_denied**
- **errors.invalid_file_access**

An example of using package **file_io** with procedure **open** follows.

```
-- Open the file "TEST.OUT" in the current directory with
-- the default settings.
--
with file_io, errors;
use  file_io;
procedure fio_test4 is

  test_handle : file_handle;
  open_error  : errors.extended_errors;
begin
  open ("Test.Out", test_handle, open_error);
end;
```

12.5 Function Close

Function **close** closes a file whose handle was previously returned by a successful call to **open**, **create**, **create_temporary**, or **create_new**.

```
function close (handle : file_handle)
    return errors.extended_errors;
```

Closing flushes the file's internal buffers to disk and releases the handle for reuse.

The file's date and time is updated if the file was modified.

This call corresponds to Interrupt 16#21#, function 16#3E#.

A possible error follows.

- `errors.invalid_handle`

An example follows.

```
-- Close the file associated with the specified handle
-- and notify user if in error.
--
with file_io, errors;
use  file_io, errors;
procedure fio_test5 (test_handle : file_handle) is

begin
  if close (handle => test_handle) /= ok then
     text_io.put_line ("Error on Close");
  end if;
end;
```

12.6 Function Cooked

Function **cooked** sets the **cooked** bit in the device driver information word for the specified standard device.

```
function cooked (for_standard : standard_io)
    return errors.extended_errors;
```

Characters that require special action are checked on input or output.

Character devices perform their input and output in the cooked mode by default.

This call corresponds to Interrupt 16#21#, function 16#44#.

12.7 Function Raw

Function **raw** sets the **raw** bit in the device driver information word for the specified standard device.

```
function raw (for_standard : standard_io)
    return errors.extended_errors;
```

Checking for Control–C (^C) or any other control characters is turned off. This increases display speed. The system does not take any action on special characters in the input stream.

This call corresponds to Interrupt 16#21#, function 16#44#.

12.8 Procedure Read

Procedure **read** reads from a file or device associated with the specified handle.

```
procedure read (
    handle         : file_handle;
    bytes_to_read  : natural;
    buffer_address : system.address;
    bytes_read     : out natural;
    error          : out errors.extended_errors
  );
```

The number of bytes actually read is returned in *bytes_read*. This value may be less than the value specified in *bytes_to_read* if:

- an error occurs.
- the end of file is encountered.
- an end–of–line sequence (carriage–return/line–feed) is read from a character device that is in cooked mode. The CR/LF pair is read into the buffer and is counted.

The read buffer associated with the *buffer_address* must be large enough for the number of bytes specified for reading.

The file position pointer is updated.

This call corresponds to Interrupt 16#21#, function 16#3F#.

Possible errors are:

- **errors.access_denied**
- **errors.invalid_handle**

An example of using package **file_io** with procedure **read** follows.

```
-- Read a maximum of 80 bytes or until a carriage-return
-- is detected from the standard input device.
--
with file_io, errors;
procedure fio_test6 is

   buffer       : string (1 .. 80);
   qty_read     : natural;
   read_error   : errors.extended_errors;
begin
   file_io.read (
       handle         => stdin,
       bytes_to_read  => 80,
       buffer_address => buffer (1)'address,
       bytes_read     => qty_read,
       error          => read_error
     );
end;
```

12.9 Procedure Write

Procedure **write** writes to a file or device associated with the specified handle.

```
procedure write (
    handle         : file_handle;
    bytes_to_write : natural;
    buffer_address : system.address;
    bytes_written  : out natural;
    error          : out errors.extended_errors
  );
```

The number of bytes actually written is returned in *bytes_written*. This value may be less than the number specified in *bytes_to_write* if an error occurs (e.g. the disk is full).

The file position pointer is updated.

This call corresponds to Interrupt 16#21#, function 16#40#.

Possible errors are:

- `errors.access_denied`
- `errors.invalid _handle`

An example of using package `file_io` with procedure `write` follows.

```
-- Write the test string to a previously-opened file.
--
with file_io, errors;
use  file_io;
procedure fio_test7 (open_handle : file_handle) is

   qty_written  : natural;
   buffer       : string (1 .. 14) := "This is a Test";
   write_error  : errors.extended_errors;
begin
   write (
       handle          => open_handle,
       bytes_to_write  => buffer'length,
       buffer_address  => buffer (1)'address,
       bytes_written   => qty_written,
       error           => write_error
     );
end;
```

12.10 Function Delete

Function `delete` deletes a file.

```
function delete (name : string) return errors.extended_errors;
```

Wildcard characters are not valid.

Files that have the *read_only* attribute set can not be deleted until cleared by a call to *set_attributes*.

This call corresponds to Interrupt 16#21#, function 16#41#.

Possible errors are:

- `errors.file_not_found`
- `errors.access_denied`

12.11 Function Rename

Function `rename` renames a file or moves the directory entry of a file to another directory.

```
function rename (
    old_name : string;
    new_name : string
  ) return errors.extended_errors;
```

Wildcard characters are not valid.

This call corresponds to Interrupt 16#21#, function 16#56#.

Possible errors are:

- **errors.file_not_found**
- **errors.path_not_found**
- **errors.access_denied**
- **errors.not_same_device**

An example of using package **file_io** with function **rename** follows.

```
-- Move "file.dat" to directory "test2" and rename
-- it to "old.dat".
--
with file_io, errors;
use file_io;
procedure fio_test8 is

  rename_error : errors.extended_errors;
begin
  rename_error :=
    rename (
       old_name => "C:\TEST.DIR\FILE.DAT",
       new_name => "C:\TEST2.DIR\OLD.DAT"
      );
end;
```

12.12 Procedure Move_File_Pointer

Procedure **move_file_pointer** sets the file position for the next input–output operation. The position may be set to the start of the file, to the end of the file, or to a location relative to the current file position.

```
procedure move_file_pointer (
    handle         : file_handle;
    offset_method  : method_code;
    offset_value   : long_integer;
    new_pointer    : out long_integer;
    error          : out errors.extended_errors
  );
```

The value returned in *new_pointer* is the byte offset from the beginning of the file.

It is possible to set the file pointer to a location before the start of the file. If this is done, any attempt to read or write to the file causes an error. If the file pointer is positioned beyond the end of the file, a subsequent write to the file inserts "padding" (garbage characters) between the previous end of file and the selected file position.

This call corresponds to Interrupt 16#21#, function 16#42#.

Possible errors are:

- **errors.invalid_function**
- **errors.invalid_handle**

An example of using package **file_io** with procedure **move_file_pointer** follows.

```
-- Move the file pointer to the end of the file,
-- and display that position.
--
with file_io, errors, text_io;
use  file_io;
procedure fio_test9 (open_handle : in file_handle) is
  current_end : method_code;
  file_size   : long_integer;
  move_error  : errors.extended_errors;
begin
  move_file_pointer (
      handle        => open_handle,
      offset_method => current_end,
      offset_value  => 0,
      new_pointer   => file_size,
      error         => move_error
    );
   text_io.put_line (long_integer'image (file_size));
end;
```

12.13 Procedure Get_Attributes

Procedure **get_attributes** obtains the attributes of a file in the current or specified directory.

```
procedure get_attributes (
    name          : string;
    attributes    : out file_attributes;
    error         : out errors.extended_errors
  );
```

This call corresponds to Interrupt 16#21#, function 16#43#.

Possible errors are:

- **errors.file_not_found**
- **errors.path_not_found**

An example of using package **file_io** with procedure **get_attributes** follows.

```
-- Determine if the Read_Attribute is set for the file "TEST.DAT".
--
with file_io, errors;
use  file_io, errors;
procedure fio_test10 is
```

```
      attr  : file_io.file_attributes;
      error : errors.extended_errors;
   begin
      get_attributes ("TEST.DAT", attr, error);
      if error = ok and then attr (read_only) then
         -- user defined.
         --
         null;
      end if;
   end;
```

12.14 Function Set_Attributes

Function **set_attributes** sets the attributes on a file in the current of specified directory.

```
function set_attributes (
    name       : string;
    attributes : file_attributes
  ) return errors.extended_errors;
```

The **volume_label** and **subdirectory** attributes cannot be set.

This call corresponds to Interrupt 16#21#, function 16#43#.

Possible errors are:

- **errors.file_not_found**
- **errors.path_not_found**
- **errors.access_denied**

12.15 Function Set_File_Time

Function **set_file_time** modifies the time and date of a file associated with a specified handle.

```
function set_file_time (
    handle : file_handle;
    time   : timestamp_data
  ) return errors.extended_errors;
```

This call corresponds to Interrupt 16#21#, function 16#57#.

A possible error follows.

- **errors.invalid_handle**

12.16 Procedure Get_File_Time

Procedure **get_file_time** obtains the time and date of a file associated with a specified handle.

```
procedure get_file_time (
    handle : file_handle;
    time   : out timestamp_data;
    error  : out errors.extended_errors
  );
```

This call corresponds to Interrupt 16#21#, function 16#57#.

A possible error follows.

- `errors.invalid_handle`

12.17 Procedure Dup_FileHandle

Procedure `dup_filehandle` duplicates a currently open handle.

```
procedure dup_filehandle (
    handle     : file_handle;
    new_handle : out file_handle;
    error      : out errors.extended_errors
);
```

The *new_handle* refers to the same file or device at the same position as the original handle.

If the file position pointer is moved by one handle, the other handle is also moved.

This call corresponds to Interrupt 16#21#, function 16#45#.

Possible errors are:

- `errors.no_handle_available`
- `errors.invalid_handle`

12.18 Procedure Cdup_FileHandle

Procedure `cdup_filehandle` duplicates a file handle.

```
procedure cdup_filehandle (
    handle1    : file_handle;
    handle2    : file_handle;
    error      : out errors.extended_errors
);
```

Handle2 is made to refer to the same file or device as *handle1*.

A file previously associated with *handle2* is closed if the file was open.

If the file position pointer is moved by one handle, the other handle is also moved.

This call corresponds to Interrupt 16#21#, function 16#46#.

Possible errors are:

- `errors.no_handle_available`
- `errors.invalid_handle`

12.19 Procedure Find_First

Procedure `find_first` finds the first occurrence of *name_template* with the specified attributes in the current or specified directory.

```
procedure find_first (
    name_template     : string;
    transfer_area     : out transfer_data;
    file_info         : out file_data;
    error             : out errors.extended_errors;
    search_attributes : file_attributes :=
                          (others => false)

);
```

Name_template may contain wildcard characters.

If *search_attributes* contains **volume_label**, then only **volume_label** files are found. Other files are found if *search_attributes* specifies attribute combinations not including **volume_label**.

This call corresponds to Interrupt 16#21#, function 16#4E#.

Possible errors are:

- **errors.path_not_found**
- **errors.no_files**

For an example see procedure **find_next**.

12.20 Procedure Find_Next

Procedure **find_next** finds the next occurrence of the *name_template* specified in the call to **find_first**.

```
procedure find_next (
    transfer_area : in out transfer_data;
    file_info     : out file_data;
    error         : out errors.extended_errors
  );
```

Transfer_area must contain the information set there by a previous call to **find_first** or **find_next**.

A possible error follows.

- **errors.no_more_files**

An example of using package **file_io** with procedure **find_next** follows.

```
--
-- General routine to display files on disk.
--
with file_io, errors, text_io;
use  file_io, errors;
procedure display_files (template : string) is
  search_data      : transfer_data;
  file_information : file_data;
  find_error       : errors.extended_errors;
begin
  find_first (
      name_template => template,
      transfer_area => search_data,
      file_info     => file_information,
      error         => find_error
    );
  if find_error = ok then
    text_io.put_line (file_information.name);
    loop
      find_next (
          transfer_area => search_data,
          file_info     => file_information,
          error         => find_error
        );
```

```
        if find_error = ok then
          text_io.put_line (file_information.name);
        else
          exit;
        end if;
      end loop;
    end if;
  end;
```

Chapter 13 Package Interrupt

Package **interrupt** allows calls to the interrupt vectors. Refer to your technical reference manuals for complete details of these interrupts and the required parameters.

SPECIFICATION

```
package interrupt is

   type interrupt_range is
     new integer range 0 .. 16#ff#;

   type registers is
      record
       ax     : integer;
       bx     : integer;
       cx     : integer;
       dx     : integer;
       ds     : integer;
       es     : integer;
       si     : integer;
       di     : integer;
       carry  : integer; -- 0 or 1
       flags  : integer;
      end record;

    procedure vector (
        on             : interrupt_range;
        register_block : in out registers
      );

end interrupt;
```

13.1 Procedure Vector

Procedure **vector** takes an interrupt number and a list of register values. It performs the specified interrupt and returns the states of the registers following the interrupt. The CS, SS, SP, and BP registers may not be changed through this call.

```
procedure vector (
    on             : interrupt_range;
    register_block : in out registers
  );
```

Only those registers actually used by the interrupt need to be set. The **flags** component of the **registers** record is arranged as:

Bit	Description	Bit	Description
0	CF	8	TF
1	undefined	9	IF
2	PF	A	DF
3	undefined	B	OF
4	AF	C,D	IOPL*
5	undefined	E	NT*
6	ZF	F	undefined
7	SF		

*80286 flag

Note that the carry flag is provided separately, since it is the flag most often used to indicate return status in DOS calls.

Warning: Incorrect use of any interrupt may have unpredictable or disasterous results.

An example of using package **interrupt** with procedure **vector** follows.

```
with interrupt;  use interrupt;
procedure outchar(c: character) is
  reg_file: registers;
begin
  reg_file.ax := 16#0200#;
    -- Note: Sets AH = 2. DOS function for Character Output.
  reg_file.dx := character'pos(c);
  vector(
      on => 16#21#,      -- DOS interrupt
      register_block => reg_file
    );
end outchar;
```

This example shows how to use **interrupt.vector** to call a DOS function. Note that this particular DOS call is also available as **tty.put**.

Chapter 14 Package Memory

Package **memory** provides functions to allocate and deallocate memory. Functions are also present to manipulate allocated memory areas and to obtain information about available memory resources.

Note that paragraphs are 16–byte quantities.

SPECIFICATION

```
with system,
     errors;

package memory is

   type memory_segment is new integer;
   type segment_offset is new integer;

   function make (
       segment : memory_segment;
       offset  : segment_offset
     ) return system.address;

   procedure split (
       dos_address : system.address;
       segment     : out memory_segment;
       offset      : out segment_offset
     );

   procedure allocate (
       paragraphs_requested : natural;
       segment              : out memory_segment;
       largest_block_avail  : out natural;
       error                : out errors.extended_errors
     );

   function release (segment : memory_segment)
       return errors.extended_errors;

   procedure modify (
       paragraphs_requested : natural;
       segment              : memory_segment;
       largest_block_avail  : out natural;
       error                : out errors.extended_errors
     );

   function installed return integer; -- in K-bytes

   function used return natural; -- in Paragraphs

end memory;
```

14.1 Function Make

Function **make** returns an address from the specified *segment* and *offset*.

```
function make (
    segment : memory_segment;
    offset  : segment_offset
  ) return system.address;
```

This does not correspond to a particular DOS function.

14.2 Procedure Split

Procedure **split** obtains the *segment* and *offset* from a *dos_address*.

```
procedure split (
    dos_address : system.address;
    segment     : out memory_segment;
    offset      : out segment_offset
  );
```

This does not correspond to a particular DOS function.

14.3 Procedure Allocate

Procedure **allocate** dynamically allocates memory of the number of *paragraphs_requested* and returns the *segment* paragraph of the allocated memory block.

```
procedure allocate (
    paragraphs_requested : natural;
    segment              : out memory_segment;
    largest_block_avail  : out natural;
    error                : out errors.extended_errors
  );
```

If the procedure fails to allocate the amount of memory requested, then *largest_block_avail* contains the size of the largest available memory block.

A paragraph is 16 bytes in size.

This call corresponds to Interrupt 16#21#, function 16#48#.

Possible errors are:

- **errors.memory_blocks_destroyed**
- **errors.insufficient_memory**

An example of using package **memory** with procedure **allocate** follows.

```
-- Allocate a block of 64k
--
with memory, errors, system, tty;
use  memory, errors;
procedure allocate_64k is
```

```
  block_segment     : memory_segment;
  biggest_block     : natural;
  allocate_error    : errors.extended_errors;
  starting_address  : system.address;
begin
  allocate (
      paragraphs_requested => 4096,
      segment              => block_segment,
      largest_block_avail  => biggest_block,
      error                => allocate_error
    );
  if allocate_error = ok then
     starting_address :=
       make (
       segment => block_segment,
       offset  => 0
       );
  else
     tty.put ("Error is : " &
       errors.extended_errors'image (allocate_error));
  end if;
end;
```

14.4 Function Release

Function **release** releases a memory block that was previously allocated by **allocate**.

```
function release (segment : memory_segment)
    return errors.extended_errors;
```

The *segment* to release must be the same as the segment obtained from **allocate**. An attempt to release an invalid segment has unpredictable results.

This call corresponds to Interrupt 16#21#, function 16#49#.

Possible errors are:

- **errors.memory_blocks_destroyed**
- **errors.invalid_memory_block**

14.5 Procedure Modify

Procedure **modify** dynamically increases or decreases the size of a memory block that was previously allocated by **allocate**.

```
procedure modify (
    paragraphs_requested : natural;
    segment              : memory_segment;
    largest_block_avail  : out natural;
    error                : out errors.extended_errors
  );
```

If the procedure fails to modify the size of the specified *segment*, then *largest_block_avail* contains the size of the largest available block.

This call corresponds to Interrupt 16#21#, function 16#4A#.

Possible errors are:

- `errors.memory_blocks_destroyed`
- `errors.insufficient_memory`
- `errors.invalid_memory_block`

14.6 Function Installed

Function **installed** returns the number of kilobytes of memory currently installed.

```
function installed return integer; -- in K-bytes
```

This call corresponds to Interrupt 16#12#.

An example of using package **memory** with function **installed** follows.

```
-- get the total installed memory.
--
with memory, tty;
procedure show_total is

begin
   tty.put ("Total Memory is : " &
       integer'image (memory.installed));
end;
```

14.7 Function Used

Function **used** returns the size of the program at startup, excluding memory allocated during the program's execution.

```
function used return natural; -- in Paragraphs
```

The value returned is in terms of paragraphs.

The value returned can be used with `program_control.resident_quit`.

Information about the amount of memory used is obtained from the Program Segment Prefix; there is no corresponding system or BIOS call.

Chapter 15 Package Port

SPECIFICATION

```
package port is
   function  in_word (port_number : integer)
      return integer;
   function  in_byte (port_number : integer)
      return integer;
   procedure out_word (
       port_number : integer;
       data        : integer
     );
   procedure out_byte (
       port_number : integer;
       data        : integer
     );
end port;
```

USAGE

Package **port** allows byte or word input and output to the specified port.

Since the actual devices corresponding to particular 80x86 I/O ports may vary from machine to machine, you should consult the hardware reference materials or BIOS listing for your system in order to find out, for example, which I/O port corresponds to a serial chip or a speaker.

The document describing the hardware and BIOS for the IBM PC/AT is: *Personal Computer AT Technical Reference*, IBM document number 6280070.

15.1 Function In_Word

Function **in_word** transfers a word from the specified input port.

```
function  in_word (port_number : integer)
   return integer;
```

This does not correspond to a particular DOS function; it uses the 80x86 IN instruction.

15.2 Function In_Byte

Function **in_byte** transfers a byte from the specified input port.

```
function  in_byte (port_number : integer)
   return integer;
```

This does not correspond to a particular DOS function; it uses the 80x86 IN instruction.

15.3 Procedure Out_Word

procedure **out_word** transfers a word to the specified output port.

```
procedure out_word (
    port_number : integer;
    data        : integer
  );
```

This does not correspond to a particular DOS function; it uses the 80x86 OUT instruction.

15.4 Procedure Out_Byte

Procedure **out_byte** transfers a byte to the specified output port.

```
procedure out_byte (
    port_number : integer;
    data        : integer
  );
```

This does not correspond to a particular DOS function; it uses the 80x86 OUT instruction.

Chapter 16 Package Program_Control

Package **program_control** provides functions that exert control over the executing program, that execute other programs, or that obtain information about resident programs.

SPECIFICATION

```
with errors,
     memory;

package program_control is

   subtype byte is integer range 0 .. 255;

   type program_end is (
       voluntary_end, ctrl_break_end,
       device_error,  stay_resident_end
     );

   type state_type is (off, on);

   procedure quit (return_code : byte);

   function  resident_quit (
       return_code         : byte;
       reserve_paragraphs : natural
     ) return errors.extended_errors;

   procedure get_return_code (
       return_code  : out byte;
       how_ended    : out program_end
     );

   function  break_status return state_type;
   procedure set_break_status (to_state : state_type);

   function segment_prefix return memory.memory_segment;

   function execute (
       program_name        : string;
       command_arguments   : string
     ) return errors.extended_errors;

   function msdos (command_line : string)
       return errors.extended_errors;

   procedure get_environment_variable(
       variable_name : string;
       value         : out string;
       last          : out natural
     );

end program_control;
```

16.1 Procedure Quit

Procedure **quit** terminates the calling program and returns control to PC–DOS or the parent program, yielding a *return_code.*

```
procedure quit (return_code : byte);
```

The return code can be examined with the batch command **errorlevel** or by the parent program using **get_return_code.**

This call corresponds to Interrupt 16#21#, function 16#4C#.

16.2 Function Resident_Quit

Function **resident_quit** terminates the calling program without releasing its memory. Control returns back to PC– DOS or the parent program, yielding a return code.

```
function  resident_quit (
    return_code         : byte;
    reserve_paragraphs : natural
  ) return errors.extended_errors;
```

Reserve_paragraphs is the number of 16 byte units that are to remain resident. An appropriate value to use for *reserve_paragraphs* may be obtained by using the function **memory.used.**

This call corresponds to Interrupt 16#21#, function 16#31#.

16.3 Procedure Get_Return_Code

Procedure **get_return_code** obtains the return code of a child program after successful completion of the **execute** procedure.

```
procedure get_return_code (
    return_code  : out byte;
    how_ended    : out program_end
  );
```

The return code can be obtained only once using this call.

This call corresponds to Interrupt 16#21#, function 16#4D#.

16.4 Function Break_Status

Function **break_status** determines the current state of the operating system's Ctrl–Break checking flag.

```
function  break_status return state_type;
```

This call corresponds to Interrupt 16#21#, function 16#33#.

16.5 Procedure Set_Break_Status

Procedure **set_break_status** sets the current state of the operating system's Ctrl–Break checking flag.

```
procedure set_break_status (to_state : state_type);
```

This call corresponds to Interrupt 16#21#, function 16#33#.

16.6 Function Segment_Prefix

Function **segment_prefix** returns the memory segment of the calling program's program segment prefix (PSP).

```
function segment_prefix return memory.memory_segment;
```

This does not correspond to a particular DOS function.

16.7 Function Execute

Function **execute** executes the program with the specified *program_name*, passing the specified *command_arguments* to the program. The calling program is suspended during execution of the child program.

```
function execute (
    program_name        : string;
    command_arguments   : string
  ) return errors.extended_errors;
```

Upon completion of the child program, control returns back to the calling program. It is then possible to determine how the child ended by calling the **get_return_code** procedure.

This call corresponds to Interrupt 16#21#, function 16#4B#.

Possible errors are:

- **errors.file_not_found**
- **errors.access_denied**
- **errors.insufficient_memory**
- **errors.invalid_set_strings**
- **errors.invalid_format**

16.8 Function Msdos

Function **msdos** executes the PC–DOS command line interpreter with the specified command line.

```
function msdos (command_line : string)
    return errors.extended_errors;
```

If the specified *command_line* is an empty string (" ") then a new interactive command line interpreter is invoked. To return to the calling program, type the command **exit**.

This call corresponds to Interrupt 16#21#, function 16#4B#.

Possible errors are:

- **errors.file_not_found**
- **errors.access_denied**
- **errors.insufficient_memory**
- **errors.invalid_set_strings**
- **errors.invalid_format**

An example of using package **program_control** with function **msdos** follows.

```
-- Redirect directory listing to a file.
--
with program_control,
     errors,
     tty;
use  program_control;

procedure list_exe is
```

```
  status : errors.extended_errors;
begin
  status := msdos ("DIR *.EXE >EXELIST.TXT");
  if status /= ok then
     tty.put ("Error is : " &
           errors.extended_errors'image (status));
  end if;
end;
```

16.9 Procedure Get_Environment_Variable

Procedure **get_environment_variable** obtains the value associated with a particular environment variable.

```
procedure get_environment_variable(
    variable_name : string;
    value         : out string;
    last          : out natural
  );
```

The value of the environment variable, if found, is returned in *value*. The procedure raises the exception **constraint_error** if *value* is too small to hold the value of the environment variable.

Last is set to the index position of the last character in *value*. *Last* is set to zero if the specified *variable_name* is not in the environment.

Chapter 17 Package Revision

Package **revision** provides a facility for determining the revision (version) number of PC–DOS.

SPECIFICATION

```
package revision is
   incorrect_dos_version : exception;
   type number is
      record
       major : natural;
       minor : natural;
      end record;
   function dos return number;
   function dos return natural; -- major version only
end revision;
```

17.1 Exception Incorrect_Dos_Version

Exception **incorrect_dos_version** is raised when a system function is invoked that is available only on a later release of PC–DOS than is present on the system.

17.2 Function Dos

The overloaded **dos** functions are disambiguated by the return type. Either a complete revision number may be obtained in the composite type **number** or just the major revision number may be obtained.

17.2.1 Returning Complete Revision Number

The first version of function **dos** returns the complete version number of the host operating system.

```
function dos return number;
```

The version number is obtained during package elaboration through interrupt 16#21#, function 16#30#. If the antiquated version 1 PC–DOS environment is detected then the exception **incorrect_dos_version** is raised.

17.2.2 Returning Major Revision Only

The second version of function **dos** returns the major version number of PC–DOS.

```
function dos return natural;     -- Major Version Only
```

The version number is obtained during package elaboration through interrupt 16#21#, function 16#30#. If the antiquated version 1 PC–DOS environment is detected then the exception **incorrect_dos_version** is raised.

An example of using package **revision** with function **dos** follows.

```
-- determine if DOS version is lower than 3.0
--
with text_io;
procedure check_revision is
begin
```

```
      if revision.dos < 3 then
         text_io.put_line ("Raising Version Exception");
         raise revision.incorrect_dos_version;
      end if;
   end;
```

Chapter 18 Package Time

Package **time** provides procedures to get and set the current system date and time.

SPECIFICATION

```
package time is
   subtype hours_range      is natural range 0 .. 23;
   subtype minutes_range    is natural range 0 .. 59;
   subtype seconds_range    is natural range 0 .. 59;
   subtype hundredths_range is natural range 0 .. 99;
   subtype months_range is positive range 1    .. 12;
   subtype days_range   is positive range 1    .. 31;
   subtype years_range  is positive range 1980 .. 2099;
   type day_of_week is (
       sunday,   monday, tuesday, wednesday,
       thursday, friday, saturday
     );
   type months is (
       january,   february, march,    april,
       may,       june,     july,     august,
       september, october,  november, december
     );
   type packet is
     record
       hours      : hours_range;
       minutes    : minutes_range;
       seconds    : seconds_range;
       hundredths : hundredths_range;
       year       : years_range;
       month      : months_range;
       day        : days_range;
     end record;
   procedure set (
       hours      : hours_range;
       minutes    : minutes_range;
       seconds    : seconds_range;
       hundredths : hundredths_range := 0
     );
   procedure get (
       hours      : out hours_range;
       minutes    : out minutes_range;
       seconds    : out seconds_range;
       hundredths : out hundredths_range
     );
```

```
    procedure set (
        month : months_range;
        day   : days_range;
        year  : years_range
      );

    procedure get (
        month     : out months_range;
        day       : out days_range;
        year      : out years_range;
        days_name : out day_of_week
      );

    function get return packet;

    type time_format is (
        long,              -- 03:00:00 AM
        military,          -- 03:00:00
        short,             -- 03:00
        none
      );

    type date_format is (
        long,              -- December 23, 1986
        month_day_year,    -- 12/23/86
        day_month_year,    -- 23-DEC-86
        none
      );

    function image (
        time               : packet;
        time_notation      : time_format := long;
        date_notation      : date_format := long
      ) return string;

end time;
```

18.1 Procedure Set

The overloaded **set** procedures are disambiguated by their parameters. The first version sets the time of day; the second version sets the date.

18.1.1 Setting Time of Day

The first version of procedure **set** sets the system clock to a specified hour, minute, second, and hundredth of second.

```
procedure set (
    hours      : hours_range;
    minutes    : minutes_range;
    seconds    : seconds_range;
    hundredths : hundredths_range := 0
  );
```

This call corresponds to Interrupt 16#21#, function 16#2D#.

An example follows.

```
-- set the system time to 10:20:30.
--
with time;
use  time;
procedure set_time is

begin
  set (hours      => 10,
       minutes    => 20,
       seconds    => 30,
       Hundredths => 0);
end;
```

18.1.2 Setting Date

The second version of procedure **set** sets the system clock to a specified date.

```
procedure set (
    month : months_range;
    day   : days_range;
    year  : years_range
  );
```

This call corresponds to Interrupt 16#21#, function 16#2B#.

An example follows.

```
-- set the system date to July 27, 1986.
with time;
use  time;
procedure set_date is

begin
  set (month => 7,
       day   => 27,
       year  => 1986);
end;
```

18.2 Procedure Get

The overloaded **get** procedures are disambiguated by their parameters and/or return type. The first version obtains the time of day; the second version obtains the date; the third version returns a record containing both time and date.

18.2.1 Getting Time of Day

The first version of procedure **get** obtains the time of day from the system clock.

```
procedure get (
    hours      : out hours_range;
    minutes    : out minutes_range;
    seconds    : out seconds_range;
    hundredths : out hundredths_range
  );
```

This call corresponds to Interrupt 16#21#, function 16#2C#.

18.2.2 Getting the Date

The second version of procedure **get** obtains the date from the system clock.

```
procedure get (
    month      : out months_range;
    day        : out days_range;
    year       : out years_range;
    days_name  : out day_of_week
  );
```

This call corresponds to Interrupt 16#21#, function 16#2A#.

An example follows.

```
-- get the current date and display the days name.
--
with text_io,
     time;
procedure display_day is
   mon     : months_range;
   day     : days_range;
   yr      : years_range;
   today   : day_of_week;
begin
   time.get (mon, day, yr, today);
   text_io.put_line (day_of_week'image (today));
end;
```

18.2.3 Getting Both Date and Time

The third version of procedure **get** obtains both the date and time from the system clock.

```
function Get return Packet;
```

This call simply merges information from the date and time **get** procedures.

```
-- Display the current date and time
-- in the form "23-DEC-86 03:00:00".
with tty,
     time;
procedure todays_date is
begin
  tty.put_line(time.image(
         time          => time.get,
         time_notation => time.military,
         date_notation => time.day_month_year
        )
      );
end todays_date;
```

18.3 Function Image

Function **image** returns a string representation of an object of type **packet** in a specified format.

```
function image (
    time              : packet;
    time_notation     : time_format := long;
    date_notation     : date_format := long
  ) return string;
```

There is no corresponding DOS function.

For an example, refer to the previous section on function **get**.

Chapter 19 Package Tty

Package **tty** provides operations on the console terminal display and keyboard.

This package can be used in place of **text_io** under some circumstances. Package **tty** links in faster than **text_io** and calls to the **tty** subprograms run faster.

SPECIFICATION

```
with common_display_types;
use  common_display_types;
package tty is

   type shift_status_record is
      record
       insert       : boolean;
       caps_lock    : boolean;
       num_lock     : boolean;
       scroll_lock  : boolean;
       alt_shift    : boolean;
       ctrl_shift   : boolean;
       left_shift   : boolean;
       right_shift  : boolean;
      end record;

   procedure clear_screen;

   procedure put (char : character);
   procedure put (str  :  string);

   procedure put (
       row             : row_range;
       column          : column_range;
       item            : string;
       underline       : boolean    := false;
       reverse_video   : boolean    := false;
       blink           : boolean    := false;
       intensity       : boolean    := false
     );

   procedure put (
       row          : row_range;
       column       : column_range;
       item         : string;
       foreground   : color ;
       background   : background_color;
       blink        : boolean := false
     );
```

```
    procedure put (
        row              : row_range;
        column           : column_range;
        item             : character;
        underline        : boolean    := false;
        reverse_video    : boolean    := false;
        blink            : boolean    := false;
        intensity        : boolean    := false
      );
    procedure put (
        row         : row_range;
        column      : column_range;
        item        : character;
        foreground  : color;
        background  : background_color;
        blink       : boolean := false
      );
    procedure put_line (str : string);
    --
    -- Keyboard Functions
    --
    function shift_status return shift_status_record;
    function char_ready   return boolean;
    procedure get (
        scan_code : out byte;
        char      : out character
      );
    function  get (
        no_echo : boolean := false;
        direct  : boolean := false;
        clear   : boolean := false
      ) return character;
    procedure get (str : out string; last : out natural);
  end tty;
```

19.1 Procedure Clear_Screen

Procedure **clear_screen** clears the currently active display.

```
procedure clear_screen;
```

This call corresponds to Interrupt 16#10#, function 16#06#.

19.2 Procedure Put

The overloaded **put** procedures provide various means of writing to the console display.

19.2.1 Put Character

The first version of procedure **put** writes one character to the standard output device.

```
procedure put (char : character);
```

This call corresponds to Interrupt 16#21#, function 16#02#.

19.2.2 Put String

The second version of procedure **put** writes the specified string to the standard output device.

```
procedure put (str  :  string);
```

This call corresponds to Interrupt 16#21#, function 16#40#.

19.2.3 Put String, Non–Color Attributes

The third version of procedure **put** writes a string, *item*, with the specified non–color display attributes, to the specified cursor position in the currently selected page.

```
procedure put (
    row             : row_range;
    column          : column_range;
    item            : string;
    underline       : boolean     := false;
    reverse_video   : boolean     := false;
    blink           : boolean     := false;
    intensity       : boolean     := false
  );
```

If the output string reaches the end of the bottommost screen line, the screen is automatically scrolled up one line. This may destroy any statically formatted screen display.

The initial *row* and *column* position must be valid for the current display mode. If the position is invalid, the results are unpredictable.

This call corresponds to Interrupt 16#10#, function 16#0E#.

19.2.4 Put String, Color Attributes

The fourth version of procedure **put** writes a string to the specified cursor position in the currently selected page with the specified color display attributes.

```
procedure put (
    row         : row_range;
    column      : column_range;
    item        : string;
    foreground  : color ;
    background  : background_color;
    blink       : boolean := false
  );
```

If the output string reaches the end of the bottommost line, the screen is automatically scrolled up one line. This may destroy any statically formatted screen display.

The initial *row* and *column* position must be valid for the current display mode. If the position is invalid, the results are unpredictable.

This call corresponds to Interrupt 16#10#, function 16#0E#.

19.2.5 Put Character, Non–Color Attributes

The fifth version of procedure **put** writes a character to the specified cursor position in the currently selected page with the specified non–color display attributes.

```
procedure put (
    row             : row_range;
    column          : column_range;
    item            : character;
    underline       : boolean     := false;
    reverse_video   : boolean     := false;
    blink           : boolean     := false;
    intensity       : boolean     := false
  );
```

If the output character reaches the end of the bottommost line, the screen is automatically scrolled up one line. This may destroy any statically formatted screen display.

The initial *row* and *column* position must be valid for the current display mode. If the position is invalid, the results are unpredictable.

This call corresponds to Interrupt 16#10#, function 16#0E#.

19.2.6 Put Character, Color Attributes

The sixth version of procedure **put** writes a character to the specified cursor position in the currently selected page with the specified color display attributes.

```
procedure put (
    row         : row_range;
    column      : column_range;
    item        : character;
    foreground  : color;
    background  : background_color;
    blink       : boolean := false
  );
```

If the output character reaches the end of the bottommost line, the screen is automatically scrolled up one line. This may destroy any statically formatted screen display.

The initial *row* and *column* position must be valid for the current display mode. If the position is invalid, the results are unpredictable.

This call corresponds to Interrupt 16#10#, function 16#0E#.

19.3 Procedure Put_Line

Procedure **put_line** writes the specified string to the standard output device followed by a carriage–return, line–feed sequence.

```
procedure put_line (str : string);
```

This call corresponds to Interrupt 16#21, function 16#40#.

19.4 Function Shift_Status

The function **shift_status** returns the present shift status of the keyboard.

```
function shift_status return shift_status_record;
```

This status record corresponds to the ROM BIOS flag at address 0000:0417H.

This call corresponds to Interrupt 16#16#, function 16#02#.

19.5 Function Char_Ready

The function **char_ready** determines if a character is ready to be read.

```
function char_ready   return boolean;
```

This call corresponds to Interrupt 16#16#, function 16#01#.

19.6 Subprogram Get

The overloaded **get** subprograms provide various means of reading from the console keyboard.

19.6.1 Raw Get

The first version of **get** obtains the scan code and raw character from the keyboard.

```
procedure get (
    scan_code : out byte;
    char      : out character
  );
```

This call corresponds to Interrupt 16#16#, function 16#00#.

19.6.2 Get Character

The second version of **get** returns a character from the standard input device by the specified method.

```
function get (
    no_echo : boolean := false;
    direct  : boolean := false;
    clear   : boolean := false
  ) return character;
```

If the *clear* flag is set then the input buffer is first cleared before the next character is input.

If the *no_echo* and *direct* flags are both set, then Interrupt 16#21#, function 16#07# is used. Characters are read without being echoed to the standard output device. No special action is taken if a Control–C (^C) is detected.

If the *no_echo* flag alone is set then Interrupt 16#21#, function 16#08# is used. Characters are read without being echoed to the standard output device.

If the *direct* flag is set then Interrupt 16#21#, function 16#06# is used. No special action is taken if a Control–C (^C) is detected. If no character is ready for input then a null character (ASCII NUL) is returned.

19.6.3 Get String

The third version of **get** reads a string from the standard input device.

```
procedure get (str : out string; last : out natural);
```

This call performs a buffered read until a carriage return is detected.

Parameter *last* is the index position of the last valid character in the returned string.

This call corresponds to Interrupt 16#21#, function 16#0A#.

Chapter 20 Package Video

Package **video** provides various output and control functions for the Monochrome, Color Graphics, and Extended Graphics Adapter Cards.

The Color/Graphics Adapter has eight display pages (0 .. 7) in 40 column by 25 line text modes and four display pages in the 80 column by 25 line text modes. The Monochrome adapter has only one display page. Specifying an invalid display page (i.e. a display page that is not defined for the current text display mode) has unpredictable results.

Some of the overloaded subprograms are disambiguated by the type of character that they accept: either **standard.character** or **common_display_types.extended_ascii**. Be aware that the Ada pre-defined type **character** supports only seven-bit ASCII characters. If it is necessary to manipulate all eight bits of a byte, then pragma **suppress** should be used when using objects of type **character** (to avoid the exception **constraint_error**), or the type **extended_ascii** should be used.

SPECIFICATION

```
with common_display_types;
use  common_display_types;
package video is
   type video_mode is  (
       text40_bw,
       text40_co,
       text80_bw,
       text80_co,
       graphic_4_color,
       graphic_4_grey,
       graphic_bw,
       text80_bw_ma,
       graphic_color_high,
       graphic_med_ega,
       graphic_high_ega,
       graphic_extra_ega,
       color_extra_ega
     );
   type pixel_value is
      record
       xor_bit     : boolean       := false;
       pixel_color : graphic_color := background;
      end record;
   procedure set (mode : video_mode);
   procedure get_mode (
       width : out byte;
       mode  : out video_mode;
       page  : out display_page
     );
```

```
procedure get_light_pen (
    triggered : out boolean;
    pixel_row : out integer;
    pixel_col : out integer;
    char_row  : out integer;
    char_col  : out integer
  );

procedure set_active (page : display_page);
--
-- Windowing functions
--
procedure scroll_up (
    number_of_lines : byte;
    upper_row       : row_range;
    left_column     : column_range;
    lower_row       : row_range;
    right_column    : column_range;
    filler_attribute: display_attribute := (
        foreground => white,
        background => black,
        blink      => false
      )
  );

procedure scroll_down (
    number_of_lines  : byte;
    upper_row        : row_range;
    left_column      : column_range;
    lower_row        : row_range;
    right_column     : column_range;
    filler_attribute : display_attribute := (
        foreground => white,
        background => black,
        blink      => false
      )
  );

procedure clear_screen;

procedure read_char (
    item      : out extended_ascii;
    attribute : out display_attribute;
    page      : display_page := 0
  );
```

```
procedure write_char (
    item         : character;
    repeat_count : natural;
    attribute    : display_attribute := (
        foreground => white,
        background => black,
        blink      => false
      );
    page         : display_page    := 0
  );

procedure write_char (
    item         : extended_ascii;
    repeat_count : natural;
    attribute    : display_attribute := (
        foreground => white,
        background => black,
        blink      => false
      );
    page         : display_page    := 0
  );

procedure write_char (
    item         : character;
    repeat_count : natural;
    page         : display_page
  );

procedure write_char (
    item         : extended_ascii;
    repeat_count : natural;
    page         : display_page
  );

procedure set_color_palette (to_color   : color);

procedure write_tty (item : character);

procedure write_tty (item : extended_ascii);

--
-- Graphics Mode Only
--
procedure set_color_palette (
    to_palette : color_palette
  );

procedure write_tty (
    item        : character;
    which_color : pixel_value
  );
```

```
    procedure write_pixel (
        pixel_color  : pixel_value;
        pixel_row    : natural;
        pixel_column : natural
      );
    function read_pixel (
        pixel_row    : natural;
        pixel_column : natural
      ) return pixel_value;
    procedure write_graphic_char (
        item         : character;
        attribute    : pixel_value;
        repeat_count : natural
      );
end Video;
```

20.1 Type Video_Mode

The enumeration elements of **video_mode** are described in Figure 20.1.

20.2 Procedure Set

Procedure **set** sets the current video display mode.

```
procedure set (mode : video_mode);
```

This call corresponds to Interrupt 16#10#, function 16#00#.

An example follows.

```
-- set the video mode to 25x40 color text
--
video.set (mode => text40_co);
```

20.3 Procedure Get_Mode

Procedure **get_mode** obtains the current display mode, screen width, and page.

```
procedure get_mode (
    width : out byte;
    mode  : out video_mode;
    page  : out display_page
  );
```

This call corresponds to Interrupt 16#10#, function 16#0F#.

20.4 Procedure Get_Light_Pen

Procedure **get_light_pen** obtains the light pen's current status and position.

```
procedure get_light_pen (
    triggered : out boolean;
    pixel_row : out integer;
    pixel_col : out integer;
    char_row  : out integer;
    char_col  : out integer
  );
```

This call corresponds to Interrupt 16#10#, function 16#04#.

20.5 Procedure Set_Active

Procedure **set_active** sets the active video display page.

```
procedure set_active (page : display_page);
```

Valid display pages are:

- 0 .. 7 for text40_bw (mode 0), text40_co (mode 1) Color/Graphics Adapter
- 0 .. 3 for text80_bw (mode 2), text80_co (mode 3) Color/Graphics Adapter
- 0 .. 7 for text80_bw (mode 2), text80_co (mode 3) Extended Adapter

Setting an invalid page for the current mode has unpredictable results.

Procedure **set_active** does not apply to a Monochrome display.

This call corresponds to Interrupt 16#10#, function 16#05#.

An example of using package **video** with procedure **set_active** follows.

```
-- set the active display page to page 1.
--
video.set_active (page => 1);
```

20.6 Procedure Scroll_Up

Procedure **scroll_up** scrolls a specified window up and initializes the new lines with the *filler_attribute*.

```
procedure scroll_up (
    number_of_lines  : byte;
    upper_row        : row_range;
    left_column      : column_range;
    lower_row        : row_range;
    right_column     : column_range;
    filler_attribute : display_attribute := (
       foreground => white,
       background => black,
       blink      => false
      )
  );
```

If *number_of_lines* is zero then the entire window is blanked and initialized with the *filler_attribute*.

Lines scrolled above the top of the window are lost.

Scrolling is only valid for the currently selected display page.

Scrolling invalid windows has unpredictable results.

This call corresponds to Interrupt 16#10#, function 16#06#.

An example of using package **video** with procedure **scroll_up** follows.

```
-- define a window with corners 5,10 and 15,30 (row,col)
-- and scroll it up 4 lines filling in the bottom lines
-- with a blue background.
--
with video, common_display_types;
use  common_display_types;
procedure make_window is
```

```
   fill_attr : display_attribute;
begin
   video.clear_screen;
   fill_attr.background := blue;
   video.scroll_up (
       number_of_lines  => 4,
       upper_row        => 5,
       left_column      => 10,
       lower_row        => 15,
       right_column     => 30,
       filler_attribute => fill_attr
     );
end;
```

20.7 Procedure Scroll_Down

Procedure **scroll_down** scrolls a specified window down and initializes the new lines with the *filler_attribute.*

```
procedure scroll_down (
    number_of_lines  : byte;
    upper_row        : row_range;
    left_column      : column_range;
    lower_row        : row_range;
    right_column     : column_range;
    filler_attribute : display_attribute := (
        foreground => white,
        background => black,
        blink      => false
      )
  );
```

If *number_of_lines* is zero then the entire window is blanked and initialized with the *filler_attribute.*

Scrolling is only valid for the currently selected display page.

Lines scrolled below the bottom of the window are lost.

Scrolling invalid windows has unpredictable results.

This call corresponds to Interrupt 16#10#, function 16#07#.

20.8 Procedure Clear_Screen

Procedure **clear_screen** clears the currently selected display page.

```
procedure clear_screen;
```

This call corresponds to Interrupt 16#10#, function 16#06#.

An example of using package **video** with procedure **clear_screen** follows.

```
-- erase the screen.
--
video.clear_screen;
```

20.9 Procedure Read_Char

Procedure **read_char** allows eight–bit characters (bytes) to be read. Procedure **read_char** obtains the character and its attribute from the current cursor position in the specified display page.

```
procedure read_char (
    item      : out extended_ascii;
    attribute : out display_attribute;
    page      : display_page := 0
  );
```

This call corresponds to Interrupt 16#10#, function 16#08#.

An example of using package **video** with procedure **read_char** follows.

```
-- get the character and attribute from display
-- page zero.
--
declare
   char : extended_ascii;
   attr : display_attribute;
begin
   video.read_char (item => char, attribute => attr);
end;
```

20.10 Procedure Write_Char

The overloaded **write_char** procedures are distinguished by the presence or absence of the *attribute* parameter and by the type of character written.

20.10.1 Write With Attributes

The first and second versions of procedure **write_char** write a character and its attribute at the current cursor position in the specified display page.

Seven–Bit ASCII

The first version of procedure **write_char** writes a seven–bit ASCII character.

```
procedure write_char (
    item         : character;
    repeat_count : natural;
    attribute    : display_attribute := (
       foreground => white,
       background => black,
       blink      => false
      );
    page         : display_page   := 0
  );
```

The cursor position is not updated.

The results are unpredictable if the *repeat_count* causes characters to be written past the right side of the screen.

This call corresponds to Interrupt 16#10#, function 16#09#.

An example which uses package **video** with procedure **write_char** follows.

```
-- Write a string of 10 happy faces (ASCII ordinal 1) at
-- the current cursor position.
--
with video;
procedure write_happy is
begin
```

```
    video.write_char (
       item         => character'val (1),
       repeat_count => 10
      );
end;
```

Eight–Bit Characters

The second version of procedure **write_char** writes an eight–bit character.

```
procedure write_char (
    item         : extended_ascii;
    repeat_count : natural;
    attribute    : display_attribute := (
       foreground => white,
       background => black,
       blink      => false
      );
    page         : display_page    := 0
  );
```

The cursor position is not updated.

The results are unpredictable if the *repeat_count* causes characters to be written past the right side of the screen.

This call corresponds to Interrupt 16#10#, function 16#09#.

20.10.2 Write With Previous Attributes

The third and fourth versions of procedure **write_char** write the character at the current cursor position in the specified display page with the already–present attributes.

Seven–Bit ASCII

The third version of procedure **write_char** writes a seven– bit ASCII character.

```
procedure write_char (
    item         : character;
    repeat_count : natural;
    page         : display_page
  );
```

The character receives the attribute of the previous character at that position.

The cursor position is not updated.

The results are unpredictable if the *repeat_count* causes characters to be written past the right side of the screen.

This call corresponds to Interrupt 16#10#, function 16#0A#.

Eight–Bit Characters

The fourth version of procedure **write_char** writes eight–bit characters.

```
procedure write_char (
    item         : extended_ascii;
    repeat_count : natural;
    page         : display_page
  );
```

The character receives the attribute of the previous character at that position.

The cursor position is not updated.

The results are unpredictable if the *repeat_count* causes characters to be written past the right side of the screen.

This call corresponds to Interrupt 16#10#, function 16#0A#.

20.11 Procedure Set_Color_Palette

The overloaded **set_color_palette** procedures are distinguished by a color parameter appropriate to the display device.

20.11.1 Text or Graphics

The first version of procedure **set_color_palette** sets the background and border color in graphics modes, or sets just the border color in text modes.

```
procedure set_color_palette (to_color   : color);
```

This call corresponds to Interrupt 16#10#, function 16#0B#.

An example which uses package **video** with procedure **set_color_palatte** follows.

```
-- set the text modes border color to blue.
--
video.set_color_palette (to_color => blue);
    -- Note: must already be in a text mode.
```

20.11.2 CGA

The second version of procedure **set_color_palette** sets the specified palette for the Color/Graphics Adapter.

```
procedure set_color_palette (to_palette : color_palette);
```

This call is valid only in **Graphic_4_Color** mode.

This table indicates what colors are presented:

Color_Pallette	Graphic_Color
0	Background = Same as current Color1 = Green Color2 = Red Color3 = Brown
1	Background = Same as current Color1 = Cyan Color2 = Magenta Color3 = White

This call corresponds to Interrupt 16#10#, function 16#0B#.

An example follows.

```
-- set the color palette to palette 1.
--
video.set_color_palette (to_palette => 1);
```

20.12 Procedure Write_Tty

The overloaded **write_tty** procedures are disambiguated by parameters appropriate to the output device.

20.12.1 Write To Current Display Page

The first and second versions of procedure **write_tty** write a character to the currently active display page and updates the cursor position.

Seven–Bit ASCII

The first version of procedure **write_tty** writes seven–bit ASCII characters.

```
procedure write_tty (item : character);
```

This call should be made in text modes only.

Appropriate action is taken when these special characters are written:

- bell (ASCII BEL)
- linefeed (ASCII LF)
- carriage return (ASCII CR)
- backspace (ASCII BS)

The display attribute is taken from the previous character at that location.

If the cursor reaches the end of the bottommost screen line, the screen automatically scrolls up and the cursor is positioned at the "next" line.

This call corresponds to Interrupt 16#10#, function 16#0E#.

An example which uses package **video** with procedure **write–tty** follows.

```
-- write out the the string "this is a test"
--
with video;
procedure write_tty_test is

   str : string (1 .. 14) := "this is a test";
begin
   for i in str'range loop
      video.write_tty (item => str (i));
   end loop;
end;
```

Eight–Bit Characters

The second version of procedure **write_tty** writes eight–bit characters.

```
procedure write_tty (item : extended_ascii);
```

Appropriate action is taken when these special characters are written:

- bell (ASCII BEL)
- linefeed (ASCII LF)
- carriage return (ASCII CR)

- backspace (ASCII BS)

The display attribute is taken from the previous character at that location.

If the cursor reaches the end of the bottommost screen line, the screen automatically scrolls up and the cursor is positioned at the "next" line.

This call corresponds to Interrupt 16#10#, function 16#0E#.

An example follows.

```
-- write out the the string "this is a test"
--
with video;
procedure write_tty_test2 is
   str : string (1 .. 14) := "this is a test";
begin
   for i in str'range loop
      video.write_tty (item => character'pos (str (i)));
   end loop;
end;
```

20.12.2 Write Graphic Character

The third version of procedure **write_tty** writes a graphic character with the specified attribute.

```
procedure write_tty (
    item        : character;
    which_color : pixel_value
  );
```

The results are unpredictable if the current display mode is not a graphic mode.

If the cursor reaches the end of the bottommost screen line, the screen automatically scrolls up and the cursor is positioned at the "next" line.

This call corresponds to Interrupt 16#10#, function 16#0E#.

20.13 Procedure Write_Pixel

Procedure **write_pixel** turns on a pixel at the specified location.

```
procedure write_pixel (
    pixel_color  : pixel_value;
    pixel_row    : natural;
    pixel_column : natural
  );
```

Valid display locations depend on the current graphic mode. The results are unpredictable for invalid coordinates.

This call corresponds to Interrupt 16#10#, function 16#0C#.

20.14 Function Read_Pixel

Function **read_pixel** obtains the *pixel_value* for the specified position.

```
function read_pixel (
    pixel_row    : natural;
    pixel_column : natural
  ) return pixel_value;
```

Valid display locations depend on the current graphic mode.

This call corresponds to Interrupt 16#10#, function 16#0C#.

20.15 Procedure Write_Graphic_Char

Procedure **write_graphic_char** writes a graphic character at the current cursor location.

```
procedure write_graphic_char (
    item         : character;
    attribute    : pixel_value;
    repeat_count : natural
  );
```

The cursor position is not updated.

This call is valid only in graphics modes.

The results are unpredictable if the *repeat_count* causes characters to be written past the right side of the screen.

This call corresponds to Interrupt 16#10#, function 16#0A#.

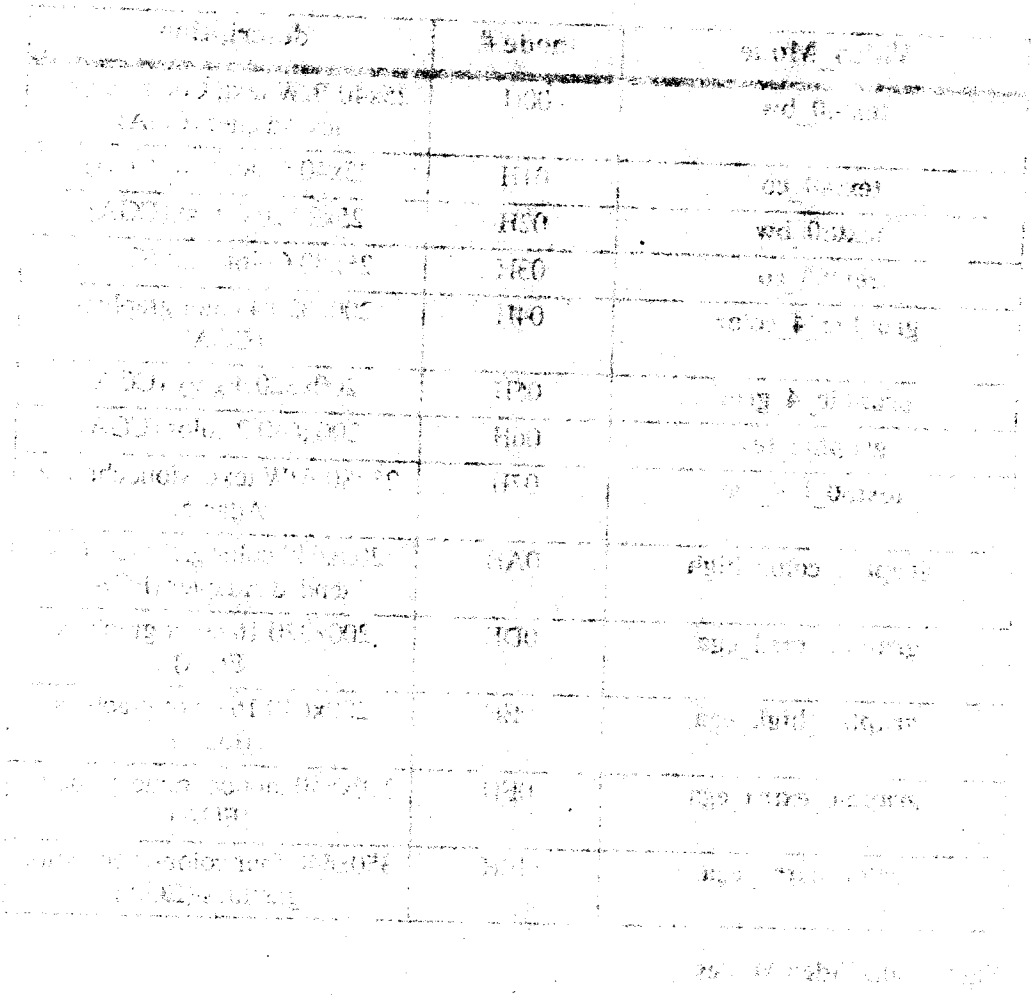
An example which uses package **video** with procedure **write_graphic_char** follows.

```
-- write the 10 graphic '?' characters,
-- wait 2 seconds then erase it.
-- (graphics mode is assumed)
--
with video, common_display_types;
use  video, common_display_types;

procedure graphic_test is
   attr : pixel_value;
begin
   attr.pixel_color := color1;
   write_graphic_char (
       item         => '?',
       attribute    => attr,
       repeat_count => 10
     );
   delay 2.0;
   attr.xor_bit := true;
   write_graphic_char (
       item         => '?',
       attribute    => attr,
       repeat_count => 10
     );
end;
```

Video_Mode	mode #	description
text40_bw	00H	25x40 B/W text, Color Graphics Adapter (CGA)
text40_co	01H	25x40 Color text (CGA)
text80_bw	02H	25x80 B/W text (CGA)
text80_co	03H	25x80 Color text (CGA)
graphic_4_color	04H	200x320 4 color graphics (CGA)
graphic_4_grey	05H	200x320 4 grey (CGA)
graphic_bw	06H	200x640 2 color (CGA)
text80_bw_ma	07H	25x80 B/W text, Monochrome Adapter
graphic_color_high	0AH	200x640 color graphics, Extended Adapter (EGA)
graphic_med_ega	0DH	200x320 16 color graphics (EGA)
graphic_high_ega	0EH	200x640 16 color graphics (EGA)
graphic_extra_ega	0FH	350x640 monochrome graphics (EGA)
color_extra_ega	10H	350x640 four color or 16–color graphics (EGA)

Figure 20.1Video Modes



Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W